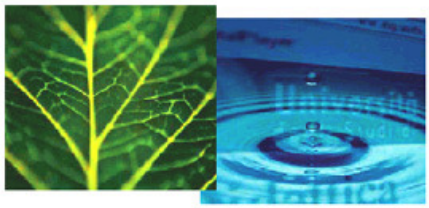
PhD Dissertation



International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Requirements-based Software System Adaptation

Vítor Estêvão Silva Souza

Advisor:

Prof. John Mylopoulos

Università degli Studi di Trento

June 2012

Abstract

Nowadays, there are more and more software systems operating in highly open, dynamic and unpredictable environments. Moreover, as technology advances, requirements for these systems become ever more ambitious. We have reached a point where system complexity and environmental uncertainty are major challenges for the Information Technology industry. A solution proposed to deal with this challenge is to make systems (self-)adaptive, meaning they would evaluate their own behavior and performance, in order to re-plan and reconfigure their operations when needed.

In order to develop an adaptive system, one needs to account for some kind of feedback loop. A feedback loop constitutes an architectural prosthetic to a system proper, introducing monitoring and adaptation functionalities to the overall system. Even if implicit or hidden in the system's architecture, adaptive systems must have a feedback loop among their components in order to evaluate their behavior and act accordingly. In this thesis, we take a Requirements Engineering perspective to the design of adaptive software systems and, given that feedback loops constitute an (architectural) solution for adaptation, we ask the question: what is the requirements problem this solution is intended to solve?

To answer this question, we define two new classes of requirements: Awareness Requirements prescribe the indicators of requirements convergence that the system must strive to achieve, whereas Evolution Requirements represent adaptation strategies in terms of changes in the requirements models themselves. Moreover, we propose that System Identification be conducted to elicit parameters and analyze how changes in these parameters affect the monitored indicators, representing such effect using differential relations.

These new elements represent the requirements for adaptation, making feedback loops a first-class citizen in the requirements specification. Not only they assist requirements engineers in the task of elicitation and communication of adaptation requirements, but with the proper machine-readable representations, they can serve as input to a framework that implements the generic functionalities of a feedback loop, reasoning about requirements at runtime. We have developed one such framework, called Zanshin, and validated our proposals through experiments based on a well-known case study adopted from the literature.

Keywords[Adaptive systems, requirements, feedback loops, awareness, evolution, system identification, *Zanshin*, *Qualia*]

Acknowledgements

This thesis would not have been possible without the help of several different people, many of whom might know even be *aware* that they have contributed to this achievement. So, other than thanking God for the amazing opportunity of coming to Trento to pursue my PhD degree, I would also like to recognize and thank colleagues, friends and family for their help during these past four years.

To my advisor, John Mylopoulos, thank you for your guidance, for always sharing with me your vast experience in the academic field, for supporting my research in every way possible. In short, thank you for helping me get to where I am now.

To Anna, Julio, Paolo and Oscar, not only for accepting the invitation to participate in my thesis committee, but for all the advice along the way, thank you very much. Big thanks also to those with whom I have worked more directly, in particular Yiqiao Wang (extending your work has earned me my first publication during the PhD), prof. William N. Robinson (for all the assistance with the monitoring framework) and Alexei Lapouchnian (for collaborating with this research from its very early stages).

To my colleagues at the University of Trento (past and present), with whom I have shared not only ideas, but also pizzas and beers, thank you for your friendship. To all the friends I have made in Trento and travelling around the world for conferences and meetings, thanks for the fun times! All of you have helped make these years seem not such a hardship.

To my family — mom, dad, my sister, my brothers, my grandmas, in-laws, uncles, aunts, cousins, nieces and nephew — thank you for always, always being there for me. To my parents, especially, I am thankful for all you have invested in me, allowing me to have reached so far.

Last, but definitely not least, thank you Renata, so much, for being the amazing wife that you are. For accompanying me to Trento, enduring some hard times with me while we adjusted to a new life. For your unconditional love and support, and for always taking care of me. As I have told you many times, this accomplishment is also yours. I love you.

To all of you, my many thanks, *grazie mille*, *muito obrigado!*

Vítor

The work compiled in this thesis has been partially supported by the ERC advanced grant 267856 "Lucretius: Foundations for Software Evolution" (unfolding during the period of April 2011 – March 2016) — `http://www.lucretius.eu.`

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The major cause [of the software crisis] is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

Edsger W. Dijkstra[†]

As technology advances, the requirements for software systems become ever more ambitious. We have reached a point where complexity is now one of the major challenges for the Information Technology (IT) industry. A solution that has been proposed by researchers in the past years is to design systems that adapt themselves to undesirable situations, such as new contexts, failures, suboptimal performance, etc. These solutions invariably include, even if hidden or implicit, some form of feedback loop, as in control systems. Only few of them, however, consider the issue of adaptation during the whole software development process, starting from requirements engineering. In this chapter, we discuss the issue of software adaptation under a requirements engineering perspective, motivated by the increased complexity of software systems and uncertainty of the environments on which they operate. At the end of this chapter, we present an overview of this thesis' proposal: a requirements-based approach for the design of adaptive systems.

1.1 Challenges of modern software systems

Nowadays, there are more and more software systems operating in highly open, dynamic and unpredictable environments, e.g., automated car driving,[1] real-time monitoring and

[†]All quotes in this thesis were taken from `http://www.wikiquote.org`.

[1]See, e.g., Google Driverless Car: `http://googleblog.blogspot.it/2010/10/what-were-driving-at.html`.

control, business analytics,[2] social networks, etc. This is pushing software to grow not only in size, but also in variability, to cope with the increasingly larger sets of requirements as a result of *environmental uncertainty* [Cheng and Atlee, 2007].

An example of the challenges ahead is the imminent need for Ultra-Large-Scale Systems [Northrop et al., 2006], such as, for instance, next-generation military command and control, future intelligent transportation management, critical infrastructure protection, integrated health-care, disaster response, etc. According to Cheng and Atlee [2007], for systems like these, requirements will come from many different stakeholders, involve multiple disciplines and be presented at varying levels of abstraction.

Another factor that has a high impact on environmental uncertainty is the increasing involvement of humans and organizations in system structures and operations, as more and more aspects of human life are being automated or assisted by computer programs. Here, there are challenges in defining system boundaries, accommodating both internal organizational rules and external laws and regulations, understanding how the technical component of the system affects the social one and vice-versa, among others [Bryl, 2009].

Lehman [1980] called these kinds of systems as *E-Type systems*. Such systems are deployed (*E* is for *Embedded*) in the real world, which has an unbounded number of properties and parameters, but are built according to a specification which is necessarily bounded, leading to the explicit or implicit inclusion of assumptions about the real world in the system specification [Lehman and Ramil, 2002]. In other words, more precisely those of Fickas and Feather [1995], "requirements are typically formulated within the context of an assumed set of resource and operating needs and capabilities. As the environment changes, it may render those assumptions invalid." Therefore, according to Lehman and Ramil [2002], it follows that this kind of system naturally obeys a principle of software uncertainty:

> "The outcome of the execution of E-type software entails a degree of uncertainty, the outcome of execution cannot be absolutely predicted", or more fully, "Even if the outcome of past execution of an E-type program has previously been admissible, the outcome of further execution is inherently uncertain; that is, the program may display inadmissible behaviour or invalid results"

This principle is also confirmed in practice. The Standish Group's 2003 CHAOS Chronicles[3] showed that, over 13 thousand IT projects (among which 15% failed and 51% overrun their time and/or cost constraints), only about half of the originally allocated requirements appear in the final released version. Because this can lead to insufficient

[2] See, e.g., `http://www-142.ibm.com/software/products/us/en/category/SWQ00`.

[3] `http://www.standishgroup.com/chaos/toc.php`.

project planning, continuous changes in the project, delays, defects, and overall customer dissatisfaction [Ebert and De Man, 2005], many approaches have been proposed to deal with this challenge, such as, for instance, agile methods [Sillitti et al., 2005], requirements definition and management techniques [Ebert and De Man, 2005], monitoring of claims over environmental assumptions [Welsh et al., 2011], among others.

The high degree of uncertainty of the environment in which a software is deployed increases its *external complexity*. On the other side of the coin, a software could also have a high degree of *internal complexity*, i.e., the problem itself being solved is a difficult one. Complexity is hard concept to define. Gell-Mann [1988] defines complexity of things in nature as the degree of difficulty in communicating them in a predefined language, comprising both how hard it is to represent them and how challenging it is to understand the theory behind the chosen language. In the case of software systems, the chosen language must be one that can be interpreted by a computer, so problems that appear to be expressed quite easily in natural language become much more complex when specified in a more formal notation, such as computer code. Moreover, problems that are currently being solved by software projects are intrinsically complex, otherwise they are not much of a problem [Hinchey and Coyle, 2012, Preface].

Internal complexity comes from the desire to automate more and more tasks in our everyday lives. The average mobile device today has one million lines of code and that number is doubling every two years; a Boeing 777 depends on 4 million lines of code whereas older planes such as the Boeing 747 had only 400 thousand; it was predicted that cars would average today about 100 million lines of code, which is ten times the amount of code they had in early 1980s [Chelf and Chou, 2008]. If this complexity continues to grow at its current rate, human intervention for system administration, maintenance, evolution, etc. may soon become infeasible [Horn, 2001].

Following Moore's law, there has been a half a million fold improvement in hardware capability in the last 30 years, which consists of an opportunity to build more and more ambitious systems. Hardware is not only becoming more powerful, but also smaller, resulting in ubiquitous microprocessors which, coupled with advances also in communication technologies, allowed for the creation of complex, globally distributed systems scaling to intercontinental distances [Butler et al., 2004]. Distributed systems tend to have global and local requirements that have to be reconciled and often change to adapt to market demands. Managing complexity in such systems has become a profitable business, as can be seen by the amount of success stories advertised by companies that are specialized in IT management.[4]

A report published by The Royal Academy of Engineering, in the United Kingdom

[4]See, for instance, CA Technologies' Success Stories: `http://www.ca.com/us/Success-Stories.aspx`.

[Butler et al., 2004] collected evidence from more than 70 individuals, encompassing senior directors, managers, project managers and software engineers from the public and private sector, as well as academic experts, concluding, among other things, that more research into complexity and associated issues is required to enable the effective development of complex, globally distributed systems. The report points out the widespread perception that the success rate of IT projects is unacceptably low, incurring losses of hundreds of billions of dollars per year in the United States and the European Union alone.[5]

One of the solutions that are being considered to the problems discussed above is more automation (which is ironic, given that the increased level of automation is one of these causes of these problems). For instance, there has been growing interest in automating activities of the software development process itself to aid in the construction of complex systems. According to a SAP security expert, the use of automated scanning of code for potential threats raised the number of security notes in their software by a huge amount in 2010, when compared to previous years, remaining as high in 2011. The reason is not that threats were not present in previous years, but that they could not be identified using solely non-automated means [Schaad, 2012]. To cite another example, a study commissioned by the National Institute of Standards and Technology (NIST) showed that around US$ 22.2 billion could be saved by an improved Verification & Validation infrastructure that enables earlier and more effective identification and removal of software defects such as, for instance, static source code analysis [Chelf and Chou, 2008].

However, the challenges illustrated in this section do not affect software-intensive systems only during their development, but also at their maintenance and evolution stages, after they have already been put into operation. In this case, it is important to build systems that can change their own behavior (possibly with humans in the loop) to continue to fulfill their requirements, despite growing environmental uncertainty and internal complexity. The next section focuses on this adaptation capability as a way of coping with uncertainty and complexity.

1.2 Software system adaptation

A solution proposed to deal with the problems discussed above is to make systems self-managed, meaning they would self-configure for operation, self-protect from attacks, self-heal from errors, self-tune for optimal performance, etc. This section discusses how (self-)adaptation can be used to manage uncertainty and complexity.

[5] We do recognize, however, there are many cases of successful IT projects involving complex, multi-million lines of code artifacts. What this section is trying to highlight is that there are still problems that need to be addressed regarding uncertainty and complexity in software projects.

1.2.1 Definitions

Researchers refer to systems with the aforementioned *self-** properties as *autonomic systems* [Kephart and Chess, 2003]. One particular self-* property that has been gaining a lot of attention from the research community is self-adaptation. In December 1997, four years before the publication of the autonomic manifesto by Horn [2001], the DARPA Broad Agency Announcement BAA-98-12 provided a definition for self-adaptive software, as quoted by Laddaga and Robertson [2004]:

> Self Adaptive Software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing 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 behavior and performance, as well as the ability to re-plan and reconfigure its operations

Many researchers use the terms *autonomic* and *self-adaptive* interchangeably. However, according to Salehie and Tahvildari [2009] there are some similarities, but also some differences between self-adaptive software and autonomic computing. In their view, the term *autonomic* refers to a broader context, handling all layers of the system's architecture (from applications to hardware), whereas *self-adaptive* has less coverage — constrained mostly to applications and middleware — and, thus, falling under the umbrella of autonomic computing.

In our view, there are also differences in the underlying motivation and the approach for solution in the research areas of autonomic computing and adaptive systems. The former, triggered by the publication of the autonomic manifesto [Horn, 2001] by IBM, is motivated by the maintenance cost of systems that have high internal complexity (e.g., compilers, database management systems, etc.), whereas the latter is fueled by the need of deploying software systems in social, open contexts with high degrees of uncertainty. Furthermore, autonomic research focuses on architectural solutions for automating maintenance tasks, whereas adaptive systems usually are more concerned with user requirements and environmental assumptions.[6] In this sense, adaptive research would have a broader scope, considering the entire software development process from requirements to operation.

In this thesis we focus on adaptation, based on the definition of self-adaptive software provided by DARPA's announcement. We do, however, provide yet another distinction,

[6] Later, in Section 2.2, we also discuss architectural approaches for adaptation. However, although not focusing on Requirements Engineering, these approaches do focus on concerns which are, in a way, user requirements, such as, for instance, quality of service.

one between the terms *adaptive* and *self-adaptive*:

- An ***adaptive*** *software* is a software system that has mechanisms for monitoring and adaptation as per DARPA's definition when complemented by external actors, such as sensors or its (human) users. An example are socio-technical systems, which include in their architecture and operation organizational and human actors along with software and hardware ones [Bryl, 2009];

- A ***self-adaptive*** *software* is a software system whose monitoring and adaptation mechanisms are fully automated, i.e., they do not involve humans in the loop when evaluating their own behavior or reconfiguring their operations. These kinds of systems are often also called *autonomous*;

In the first case, by combining an adaptive software with human and organizational actors we can form a socio-technical system that can be deemed a *self-adaptive system* given that the human and organizational actors that implement (part of) the adaptation capabilities are all included in the system as a whole. For simplicity, we henceforth refer to these software-intensive systems (socio-technical or not) as ***adaptive systems***. Note, however, that other types of systems can also be (self-)adaptive. For example, living organisms such as the human body (in effect, the autonomic nervous system that controls the human body inspired proposals included in the autonomic manifesto [Horn, 2001]).

In other words, this thesis is concerned with adaptive software-intensive systems in general, regardless whether they are self-adaptive or not. The discussions in this chapter and the concepts in subsequent chapters apply to an autonomous cleaning robot with no human intervention the same way they apply to a meeting scheduling system whose output depends on the inputs of different members of the organization. In the former, when presented with an obstacle, the robot would use its own components (e.g., actuators) to change its course and overcome the obstacle. In the latter, it could be the case that a secretary informs the system that all available rooms are full and the system decides to delegate to managers the task of increasing the number of rooms available by arranging new space for meetings.

Notice how, in the first case, monitoring and adaptation were automated by the robot, whereas in the second scenario they both involve a human in the loop — the secretary did the monitoring and the managers were in charge of the adaptation. Adaptive systems can mix both kinds of monitoring/adaptation, depending on how designers choose to implement them.

1.2.2 Feedback loops

It follows from the above that in order to develop an adaptive system, one needs to account for some kind of *feedback loop*. A feedback loop constitutes an architectural prosthetic to a system proper, introducing monitoring, diagnosis, etc. functionalities to the overall system. A well-known example is the autonomic computing MAPE loop [Kephart and Chess, 2003], whose acronym stands for the four activities that it performs: monitor, analyze, plan and execute. If adaptive systems need to evaluate their behavior and act accordingly, they must have some kind of feedback loop among their components, even if implicit or hidden in the system's architecture.

For instance, let us recall the autonomous cleaning robot and the socio-technical adaptive meeting scheduler exemplified earlier. Following the terminology used in control systems development, we refer to these systems as *target systems*. In order to know that there is an obstacle ahead or that there are no available rooms for meetings, target systems must have the ability to monitor these properties and feed them back into some decision process — sometimes referred to as the *controller* or *adaptation framework* — which will switch from the current behavior to an alternative one that overcomes the undesirable situation. For the robot this means calculating a new path of movement towards the target, whereas the meeting scheduler could decide that a new room should be made available, or that the meeting should be postponed, or yet that another meeting should be canceled in order to make a room available, etc.

Recently, researchers have expressed the need to make these feedback loops first-class citizens in the design of adaptive systems. Brun et al. [2009] notice that "while [some] research projects realized feedback systems, the actual feedback loops were hidden or abstracted. [...] With the proliferation of adaptive software systems it is imperative to develop theories, methods and tools around feedback loops." Andersson et al. [2009] consider that "a major challenge is to accommodate a systematic engineering approach that integrates both control-loop approaches with decentralized agents inspired approaches." Cheng et al. [2009b] declare that "Even though control engineering as well as feedback found in nature are not targeting software systems, mining the rich experiences of these fields and applying principles and findings to software-intensive adaptive systems is a most worthwhile and promising avenue of research for self-adaptive systems. We further strongly believe that self-adaptive systems must be based on this feedback principle."

Considering feedback loops as first class citizens implies a fundamental difference between the design of "vanilla" (i.e., non-adaptive) systems and that of adaptive systems: requirements for the latter are not necessarily treated as invariants that must always be achieved. Instead, we accept the fact that the system may fail in achieving any of its initial objectives and provide a way of specifying the level of criticality of each goal as

constraints on their success/failure and assigning adaptation actions to be carried out when the system does not fulfill these constraints.

Feedback loops are a fundamental architectural element in the design of control systems, whereby the output to be controlled is compared to a desired reference value and their difference is used to compute corrective control action [Doyle et al., 1992]. In other words, measurements of a system's output are used to achieve externally specified goals by adjusting *parameters* that in some way affect *indicators* that these goals are being achieved. For this reason, feedback loops are also called *closed loops* and are present in some form in almost any system that is considered automatic [Hellerstein et al., 2004], such as an automobile cruise control or an industrial control system of an electric power plant.

On the other hand, *open loops*, also known as *feed-forward loops*, are different in the sense that they calculate the control action without measuring the output of the system [Hellerstein et al., 2004]. Instead, they change their behavior according to information monitored exclusively from the environment in which they operate, regardles of what is the current output of the system. Context-aware systems are examples of systems that use open loops.

Control Theory, an interdisciplinary branch of engineering and mathematics, is concerned with the behavior of dynamical systems, such as the ones cited above, providing ways to determine the proper corrective action when a system's output does not match the desired, reference values. For example, one of these ways is a process known as *System Identification*, which consists of relating the adjustable parameters to the monitored indicators through *differential relations*. During the research that is compiled in this thesis, we have applied and taken inspiration from Control Theory in several occasions, in order to formulate our proposals on the design of adaptive systems.

1.2.3 Requirements-based adaptation

As we have just discussed, feedback loops provide the means through which adaptive systems are able to monitor important indicators and, if they show the target system is not working properly, take appropriate corrective action in order to adapt. Then, the questions that follow are: which are the important indicators to monitor? What are the proper corrective actions for their failure? Who decides these things? An answer to these questions has been given more than three decades ago, by Ross and Schoman Jr. [1977]:

> Requirements definition must encompass everything necessary to lay the groundwork for subsequent stages in system development. [...] It must say why a system is needed, based on current or foreseen conditions, which may be in-

> ternal operations or an external market. It must say what system features will serve and satisfy this context. And it must say how the system is to be constructed.

Like any other kind of functionality, adaptivity features should also be described in the software requirements specification in all of its aspects: why it is needed (i.e., what are the goals to satisfy), what should be developed (i.e., what features will serve and satisfy the goals) and how it should be constructed (i.e., what are the quality criteria used to measure these features). Absence of this information can lead to delays in software delivery, excessively high cost in development and ultimately dissatisfied clients [Ross and Schoman Jr., 1977].

Take, for instance, a management system for an emergency service integrating police, fire departments and emergency medical services of a given city or region. Dealing with emergencies that may threaten people's lives, this system is considered critical and, therefore, should ensure that it always satisfies stakeholder requirements. When developing such a system, one may choose among different means to certify that it will always reach its objectives, for instance:

- Perform longer and more thorough Verification & Validation during the software development process and establish strict procedures to be followed by the staff in order to attempt to avoid system failures altogether;
- Increasing available resources, such as staff and vehicles, in order to have them available when undesirable situations present themselves, which would make it easier to respond to these situations more quickly;
- Build adaptation mechanisms into the system, making it able to detect when something has failed, determine what could be done to overcome the failure and change its own behavior in order to attempt to always satisfy its main requirements.

By stating the goals to be satisfied (e.g., critical services should always work properly) and the possible different ways to achieve them (for instance, the examples above), one can choose the best solution for the problem at hand before going into the requirements for this solution, at the same time defining why the solution is needed (e.g., adaptation is needed because system failures can have disastrous results). For the past couple of decades, Requirements Engineering (RE) research has increasingly recognized the leading role played by goals in the RE process [van Lamsweerde, 2001]. For the same reasons, our research, presented herein, is based on the state-of-the-art in Goal-Oriented Requirements Engineering (GORE).

Goal-oriented, requirements-based adaptation consists, then, of combining the concepts of GORE with the principles of Feedback Control Theory. Consider a goal model that represents the requirements of a system-to-be (e.g., an $i^\star$ strategic rationale model [Yu et al., 2011] under the system's perspective), using as modeling primitives concepts such as goals, softgoals, quality constraints and domain assumptions [Jureta et al., 2008]. This model, which embodies, among other things, the system requirements, represents what the system should do and how it should behave. A feedback loop around this system could then be built in such a way that it considers the information represented in this model as the reference value for the system, activating corrective actions whenever a *requirements divergence* is detected, i.e., whenever the output indicates that the system is not behaving as stated in the model.

Hence, the challenges here are extending the current state-of-the-art in GORE with elements that allow requirements engineers to specify, preferably with different levels of criticality, which requirements in particular should be monitored by the feedback loop (in other words, which are the indicators of *requirements convergence*), what are the system parameters that can be adjusted as part of the corrective control action and, finally, what are the relations between these parameters and the chosen indicators.

Moreover, when eliciting and modeling requirements and information about the domain, in particular the aforementioned relations between parameters and indicators, it is often the case that domain experts are not able to provide the same level of precision that is expected in traditional Feedback Control Theory. We need, therefore, to be able to represent and use information in different levels of precision and, for this purpose, we could take advantage of research results in the area of qualitative representation and reasoning [Forbus, 2004].

Finally, following trends set by research agendas and workshops on the subject (e.g., [Sawyer et al., 2010; Bencomo et al., 2011a,b]), this new, augmented requirements model could be used at runtime by an adaptation framework that is able to perform the generic steps of a feedback control loop, alleviating developers of much of the effort in implementing the feedback controller from scratch. Having an online representation of the requirements model at runtime (i.e., a machine-readable representation of the system requirements that is parsed by a framework at runtime for reasoning purposes) is also important when dealing with requirements evolution, which we discuss next.

1.2.4 Adaptation and evolution

Our work is part of a broader research project called *Lucretius: Foundations for Software Evolution*,[7] which is concerned with investigating the role of requirements in the evolution

[7] http://www.lucretius.eu/.

of software systems. In particular, one of the objectives is to develop techniques for designing software systems that evolve in response to changes in their requirements and operational environment, i.e., systems that can cope with requirements evolution.

The problem of requirements evolution was initially addressed in the context of software maintenance. In that context, requirements evolution was treated as a post-implementation phenomenon (e.g., [Antón and Potts, 2001]) caused by changes in the operational environment, user requirements, operational anomalies, etc. A lot of research has been devoted to the classification of types of changing requirements such as mutable, adaptive, emergent, etc. [Harker et al., 1993] and factors leading to these changes.

In our research, we consider that system evolution can be done in one of three possible evolution modes, namely:

- **Automatic evolution:** the system monitors its own output and the environment within which it operates and adapts its behavior to ensure that it continues to fulfill its mandate. For example, suppose a meeting scheduler sends e-mail messages to people that have been invited for a meeting, asking them for their schedule in the following week before scheduling the meeting. Furthermore, imagine that there is a quality constraint that imposes a threshold on the time it takes to schedule a meeting. Then, if the system detects that it is not satisfying the constraint, it could switch to checking people's schedules in the organization's personal information management system (i.e., replace a requirement with another), because it takes much less time;

- **Manual evolution:** system developers/maintainers evolve the system by changing its requirements and domain (environment) models in accordance with external changes, and then propagate these changes to other parts of the system, such as its architecture, code and interfaces. Think of the same example as before, however the organization does not have a personal information management system and managers decide that the meeting scheduler should be upgraded (i.e., new features should be elicited, designed and implemented) in order to be able to serve also as an integrated calendar for all of its users;

- **Hybrid evolution:** the system is operating with a feedback loop as in the automatic case, but has humans in the loop to approve system-generated compensations, or even contribute to their generation, as in the manual mode. Again using the same example, suppose the solution is not to switch from e-mail to integrated calendar, but instead have secretaries call people and ask for their availability. The secretary is the human-in-the-loop in the new requirement, which replaces the old one.

Not surprisingly, the *automatic* and *hybrid* evolution modes resemble, respectively, with what we have previously defined as *self-adaptive* and *adaptive* systems. This is

a consequence of the fact that we define requirements evolution as any change in the system's original requirements, be it a foreseen change (which could be conducted in an automatic fashion) or an unforeseen one (which would have to be done manually). Given that the software evolution process is a feedback system [Lehman, 1980], some overlapping between adaptation and evolution is, after all, expected.

For example, anticipated requirement evolution could be the result of stakeholder statements such as "If we detect so many problems in satisfying requirement R, replace it with a less strict version of it, $R-$" or "Starting January 1st, 2013, replace requirement S with S' to comply with new legislation that has been recently approved". Of course, when replacing a requirement Q with a new requirement Q' automatically, both Q and Q' must have already been implemented, i.e., software engineers must have already conducted a software development process (elicitation, design, implementation, etc.) for both alternatives.

Other proposals (e.g., [Ernst et al., 2011]), however, consider as evolution only unanticipated changes that, therefore, are not able to be modeled, let alone developed, a priori. Under this assumption, one cannot consider evolution to be conducted automatically since, at least for the foreseeable future, software is not able to think and be truly intelligent and creative [Berry et al., 2005] in order to autonomously conduct a software development process for requirements that were not anticipated by the software engineers.

Hence, in the context of our work, to evolve the requirements model in an automatic or hybrid mode consists of one way of adapting to undesirable situations, once these are detected by the monitoring component of the system's feedback loop. The main difference between *evolution* and the "usual" form of adaptation — which henceforth will be referred to as *reconfiguration* — is that the former adapts through changes in the problem space (i.e., changes parts of the requirements model), whereas the latter looks for corrective actions in the solution space (i.e., changes the current values of system parameters, without modifying the requirements model). As will be described in the rest of this chapter, the approach we propose in this thesis deals with both types of adaptation.

1.3 Objectives of our research

In Section 1.2, we have outlined the context in which this thesis is inserted, i.e., requirements-based (self-)adaptation through reconfiguration or evolution, using concepts and tools from Feedback Control Theory. We now specify more explicitly our research objective and the questions that this thesis proposes to answer.

> **Research objective**: *to define a systematic process for the design of adaptive software systems based on requirements, centered on a feedback loop that per-*

forms reconfiguration or evolution in order to adapt the system to undesirable situations, represented as requirement divergences.

We decompose the above statement into the following research questions:

RQ1: What are the requirements that lead to the adaptation capabilities of a software system's feedback loop?

If feedback loops constitute an (architectural) solution for adaptation in software systems, what is the requirements problem this solution is intended to solve? We address this question by proposing new types of requirements that represent how the different parts of the feedback loop should behave:

- *Awareness Requirements* represent the requirements for the monitoring part of the loop, indicating, with different levels of criticality, which other requirements should always be satisfied (and, therefore, if they are not, some adaptation should be done);
- The process of System Identification is adapted from Control Theory, producing *differential relations* between adjustable system *parameters* to *indicators* of requirements convergence (Awareness Requirements can be used as the latter). These indicator/parameter relations represent the requirements for the adaptation (reconfiguration) part of the loop;
- Finally, *Evolution Requirements* represent the cases in which adaptation is done through specific changes in the requirements models, as illustrated in Section 1.2.4.

RQ2: How can we represent such requirements along with the system's "vanilla" requirements?

It is important not only to elicit the requirements that lead to the adaptation capabilities of the system, but also to represent them in a way they can be easily communicated to, and understood by other developers. In particular, we have to consider the issue of qualitative representation of requirements outlined in Section 1.2.3. We address this question by allowing analysts[8] to represent information such as the indicator/parameter relations that guide adaptation (mentioned in **RQ1**) in different levels of precision, ranging from highly qualitative statements such as "parameter P affects indicator I positively" to quantitative information like "changing parameter P by x will provoke a change of $1.5x$ on indicator I".

[8]In this thesis, we use the terms *analyst* and *requirements engineer* interchangeably.

Moreover, the entire requirements model should also be represented in a way that allows reasoning to be performed at runtime by the controller that operationalizes the feedback loop. This representation would include both the "vanilla" requirements plus the requirements for system adaptation. We address this by using representations that are formal enough to be interpreted by a framework but at the same time not too formal in order not to impose unnecessary burden on requirements engineers. This run-time framework is the subject of **RQ3**, next.

RQ3: How can we help software engineers and developers implement this requirements-based feedback loop?

As suggested in **RQ2**, above, and also outlined in Section 1.2.3, requirement models can be used not only as a way of communicating information among software engineers, but they can also be used as input to a run-time controller that operationalizes the feedback loop described in Sections 1.2.2 and 1.2.3. This controller would implement both types of adaptation — i.e., reconfiguration and automatic evolution — using the information described earlier in **RQ1**.

We address this question by suggesting a systematic process that starts from the requirements of a non-adaptive software system using the state-of-the-art in Requirements Engineering and builds, step by step, the specification of an adaptive system. Such specification uses the modeling elements mentioned earlier, representing the system's adaptation requirements in terms of the requirements for the feedback loop that operationalizes the adaptation.

Furthermore, we propose a framework that takes as input: (a) the models of the aforementioned specification of the adaptive system, represented in a machine-readable format (as described in **RQ2**); and (b) log information that describes the actual outcome of the running system, in order to: (1) detect when undesirable situations have occurred; (2) calculate the appropriate corrective action to each case; and (3) instruct the system on how to adapt. Our framework is generic, in the sense that it can be used to adapt any kind of system, as long as the necessary input and framework–system communication channels are provided. As a result, any adaptation action that involves application-specific tasks will still need to be implemented by the target system's development team, which is one of the main limitations of our approach (limitations are further discussed in the last chapter of the thesis).

RQ4: How well does the approach perform when applied to realistic settings?

An important aspect of any research proposal is validation. Hevner et al. [2004]

describe five categories of evaluation methods in Design Science: Observational, Analytical, Experimental, Testing and Descriptive. Methods range from simple description of scenarios up to full-fledge case studies which are conducted in business environments. In particular, we applied the following methods to the research presented in this thesis (descriptions taken from [Hevner et al., 2004]):

- Scenarios (descriptive): construct detailed scenarios around the artifact to demonstrate its utility;
- Informed argument (descriptive): use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artifact's utility;
- Controlled experiment (experimental): study artifact in controlled environment for qualities;
- Simulation (experimental): execute artifact with artificial data.

Therefore, to address this question, throughout the thesis we illustrate (i.e., use the descriptive methods listed above[9]) the different aspects of our proposal using the classic example of the Meeting Scheduler in order to facilitate the understanding of the new concepts that our approach brings to the design of adaptive systems. To take the evaluation a bit closer to real settings, however, we have conducted experimental evaluation with the development of an Adaptive Computer-aided Ambulance Dispatch (A-CAD) System, whose requirements were based on the well-known London Ambulance Service Computer-Aided Despatch (LAS-CAD) failure report [Finkelstein, 1993] and some of the publications that analyzed the case (e.g., [Kramer and Wolf, 1996]). The requirements for the A-CAD's adaptation capabilities were elicited (**RQ1**), represented in a machine-readable format (**RQ2**) and, finally, simulations of real failure scenarios were implemented in order to test the adaptation framework (**RQ3**).

We have also assisted in the orientation of a masters student that applied our approach to the design of an adaptive system based on an existing project that designed and implemented an Automatic Teller Machine (ATM) [Tallabaci, 2012]. His dissertation evaluated parts of the approach presented in this thesis and provided the initial models that can be used as the basis for an experiment in which a real system — i.e., the GUI that simulates a physical ATM that was developed

[9] We have to recognize, however, that the experiments conducted did not have the required level of formality of a controlled experiment, as described by Wohlin et al. [1999] or Easterbrook and Aranda [2006]. For this reason, we henceforth refer to our empirical research using only the word *experiments*, without the aforementioned adjective.

earlier — can be made adaptive through the elicitation of Awareness Requirements and the execution of System Identification.

Although we have not been able to conduct deeper forms of validation, such as surveys with practitioners, field experiments or case studies, we nonetheless recognize their importance, especially if one intends to take the results of this research into a more industrial setting.

In the following section we summarize our approach, showing in a little more detail how it contributes to answering the above research questions.

1.4 Overview and contributions

Our approach for the design of adaptive systems can be described in three major steps: Awareness Requirements engineering, System Identification and Evolution Requirements engineering.

1.4.1 Awareness Requirements engineering

The first step of the approach is the elicitation of Awareness Requirements (*AwReqs*), which are requirements that talk about the states assumed by other requirements — such as their success or failure — at runtime. By basing our approach on GORE, *AwReqs* can refer to:

- **A task**: to determine if the system is able to perform a specific set of actions successfully. Using the Meeting Scheduler as illustration, an example of an *AwReq* that refers to a task is "task *Have the system schedule (the meeting)* should never fail" — in other words, every time the system attempts to produce a schedule for a meeting, it should succeed;

- **A goal**: to determine if the system is able to satisfy an objective. Referring to goals or tasks allows the system to monitor for failures in functional requirements. Example: "considering one week periods, the success rate of goal *Collect timetables* should not decrease three times consecutively";

- **A quality constraint**: to determine if the system abides by the quality criteria that have been imposed on it. Referring to quality constraints allow the system to monitor for compliance to non-functional requirements. Example: "quality constraint *At least 90% of participants attend (meetings)* should have 75% success rate";

- **A domain assumption**: to determine if things that were assumed to be true for the system's proper functioning indeed are (and continue to be) true during system operation. Referring to domain assumptions allows the system to monitor for changes in its environment (i.e., to be also context-aware). Example: "domain assumption *Participants use the system calendar* should always be true".

AwReqs can also refer to other *AwReqs*, e.g., "*AwReq AR1* should succeed 90% of the time", constituting meta-*AwReqs* or "Level 2 *AwReqs*" (further levels are also possible), being monitored in the same fashion as non-meta (i.e., Level 1) *AwReqs*. Note how some *AwReqs* refer to single instances of requirements (the task and domain assumption examples), whereas others refer to the the whole requirement class in an aggregate way (goal and quality constraint examples). At runtime, the elements of the goal model are represented as classes and instances of these classes are created every time a user starts pursuing a requirement or when they are bound to be verified. Their state (*succeeded*, *failed*, etc.) is then monitored by the feedback controller.

In summary, *AwReqs* represent undesirable[10] situations to which stakeholders would like the system to adapt, in case they happen. That way, they constitute the requirements for the monitoring component of the feedback loop that implements the adaptive capabilities of the system. As outlined in Section 1.2, adaptation can then be done through reconfiguration or evolution. For the former, we conduct System Identification.

1.4.2 System Identification

AwReqs can be used as indicators of requirements convergence at runtime. If they fail, a possible adaptation strategy is to search the solution space to identify a new configuration (i.e., values for system parameters) that would improve the necessary indicators. In other words, the system would be tuned in order to attempt to avoid further *AwReq* failures.

As in control systems (cf. Section 1.2.2), to know the effect each parameter change has on indicators we conduct System Identification for the adaptive system. In some cases (e.g., a car's cruise control), and given the necessary resources, it is possible to represent the equations that govern the dynamic behavior of a system from first principles (e.g., quantitative relations between the amount of fuel injected in the engine and the velocity of the car in different circumstances). For most adaptive systems, however, such models are overly complex or even impossible to obtain. For this reason, we adopt ideas from Qualitative Reasoning (e.g., [Kuipers, 1989]) to propose a language and a systematic system identification method for adaptive software systems that can be applied at the

[10] In general, adaptive systems will adapt to situations that are undesirable. However, *AwReqs* can represent any situation that refers to the states assumed by requirements at runtime, being them undesirable or not.

requirements level, with the system not yet developed and its behavior not completely known.

For instance, suppose the example *AwReq* referring to the success rates of goal *Collect timetables* presented earlier fails at runtime. When this happens, the controller should know which parameter can be modified in order to improve the chances of success of this goal. Suppose further there is a parameter *FhM* which indicates *From how Many* participants we should contact in order to consider the goal satisfied (represented as a percentage value). In this case, decreasing the value of *FhM* could be considered an adaptation action here. Likewise, if this goal is OR-refined into tasks *Email participants*, *Call participants* and *Collect automatically from system calendar*, selecting *call* instead of *email* or *system* instead of *call* could also help. These same changes might also have impact on other indicators (e.g., the quality constraint *At least 90% of participants attend (meetings)* also mentioned earlier), therefore all available information should be taken into consideration by the controller when deciding the new configuration of the system.

As the examples illustrate, in our approach parameters can be of two flavors. *Variation points* consist of OR-refinements which are already present in high variability systems and merely need to be labeled. For instance, we could label the refinement of goal *Collect timetables* as *VP1*, with possible values *email*, *call* or *system*. *Control variables* are abstractions over large/repetitive variation points, simplifying the OR-refinements that would have to be modeled in order to represent such variability. In the case of *FhM*, which is a numeric control variable, if it were translated to a variation point, it would produce an excessively large (potentially infinite, if *FhM* were a real number) OR-refinement: *Collect timetables from 1% of participants*, *Collect timetables from 2% of participants*, and so on.

After indicators and parameters have been identified, the effects that changes on parameters have on the outcome of indicators are analyzed and finally represented using differential equations such as, e.g.:

$$\Delta\left(AR2/FhM\right) < 0 \tag{1.1}$$

$$\Delta\left(AR2/VP1\right)\{email \rightarrow call, email \rightarrow system, call \rightarrow system\} > 0 \tag{1.2}$$

Given that *AR2* is the identifier for the *AwReq* that refers to goal *Collect timetables*, Equation (1.1) states that increasing the value of *FhM* has a negative effect on the success rate of *AR2*, whereas Equation (1.2) means that performing the changes illustrated between curly brackets can have a positive effect on the same *AwReq*. Of course, the analogous opposite relations are also inferred. Further steps of the System Identification process would compare the equations, possibly indicating if a parameter is more effective than the other in improving an indicator and represent the fact, for instance, that changes in *VP1* produce greater effect on *AR2* than changes on *FhM* (as before, this information

should be elicted from stakeholders and domain experts):

$$|\Delta\left(AR1/VP1\right)| > |\Delta\left(AR2/FhM\right)| \tag{1.3}$$

Given this information, in order to "close the feedback loop" we propose a framework that parses the goal model augmented with the information elicited during System Identification, such as the ones illustrated above, and reconfigures the system in order to adapt. An important characteristic of this framework is the ability to accommodate different levels of precision of the qualitative information between parameters and indicators. To accomplish that, the framework was made extensible, allowing for different adaptation algorithms to be executed, depending on the availability and precision level of the information.

This need comes from the fact that during requirement engineering, only some knowledge about the behavior of the system might be available initially, with more information being uncovered in time. For instance, some indicators could have all of their relations ordered by magnitude of the effect (e.g., as in Equation (1.3) above), whereas others would have just some ordering or no ordering at all. Our proposed language (illustrated in the examples above) also supports going from qualitative (e.g., $\Delta\left(I/P\right) > 0$) to quantitative (e.g., $\Delta\left(I/P\right) = 1.5$) representations. The framework itself can have an important role in eliciting, improving or even correcting information from System Identification by analyzing the history of failures, reconfigurations and their outcome in practice (although this is not implemented in our current prototypes).

1.4.3 Evolution Requirements engineering

As illustrated earlier, it is often the case that the requirements elicited from stakeholders for a system-to-be are not carved in stone, never to change during the system's lifetime. Rather, stakeholders will often hedge with statements such as "If we detect so many problems in satisfying requirement R, replace it with a less strict version of it, $R-$" or "Starting January 1$^{\text{st}}$, 2013, replace requirement S with S' to comply with new legislation that has been recently approved". Because they prescribe desired evolutions for other requirements, we refer to this kind of statement as *requirement evolution requirements*, *Evolution Requirements* or simply *EvoReqs*.

Although there are many potential benefits and uses for a systematic representation of requirements of this sort in system models, in our approach we concentrate on the application of *EvoReqs* to the design of adaptive systems. In other words, we use them to perform evolution of the model in an automatic or hybrid way — as outlined in Section 1.2.4 — in response to undesirable situations, i.e., system failures. Like the qualitative reconfiguration process described in the previous section, such evolution actions would also be

conducted by the feedback loop controller, in response to *AwReq* failures.

EvoReqs are specified using a set of primitive operations to be performed over the elements of the model. Each operation is associated with application-specific actions to be implemented in the system. Furthermore, they can be combined using patterns in order to compose *adaptation strategies*, such as "Retry", "Delegate", "Relax", etc. A simple example from the Meeting Scheduler could be attached to an *AwReq* that imposes that "task *Characterize meeting* should never fail" in order to indicate that, if this *AwReq* fails, the meeting organizer should simply retry the task in 5 seconds. The specification for this *EvoReq* is shown in Listing 1.1.

Listing 1.1: Specification of an *EvoReq* that retries task *Characterize meeting* after 5 seconds.

```
1 t' = new-instance(T_CharacMeet);
2 copy-data(t, t');
3 terminate(t);
4 rollback(t);
5 wait(5s);
6 initiate(t');
```

As the listing shows, the *EvoReq* is specified as a sequence of operations, in this case on a particular instance of a requirement that failed. Every time someone uses the system to schedule a meeting, requirements instances are created, in the same fashion as described in Section 1.4.1. In the example, if an instance of the task *Characterize meeting* fails, create a new instance of the same task, copy session data from the failed instance to the new one, terminate and rollback the failed instance, wait 5 seconds and initiate the new, undecided instance.

Each operation has a specific meaning for the controller (e.g., `new-instance()` tells the controller to create a new run-time representation of a requirement) and/or for the Meeting Scheduler (e.g., `rollback()` tells the system to undo any operations that were performed before the failure that might leave the system in an inconsistent state). At runtime, an Event-Condition-Action-based process uses the information expressed by *EvoReqs* in order to direct the system on how to adapt. This process coordinates possible different applicable strategies, choosing which one to apply and checking if the problem they attempt to remedy has been solved.

1.4.4 Contributions

In summary, the contributions of this thesis are:

- New types of requirements — *AwReqs* and *EvoReqs* — and properties of the system and its domain — parameters that can be adjusted at runtime and differential relations between these parameters and *AwReqs* (representing indicators of requirements convergence). These new model elements constitute the requirements for a feedback

control loop that monitors and adapts (through reconfiguration or evolution) the system. Ergo, this contribution addresses research question **RQ1**;

- Representation of these requirements in languages more formal than natural language in order to promote a clear and unambiguous way of describing them, addressing **RQ2**. Furthermore, these representations are adapted whenever necessary in order to make them machine-readable, which contributes to **RQ3**;

- A systematic process for conducting System Identification, including heuristics on how to identify indicators, parameters and relations between them, further refining these relations in a final step. This also contributes to **RQ3**;

- A framework that operationalizes the generic operations of the feedback loop and, therefore, capable of augmenting systems with adaptation capabilities, provided these systems satisfy some prerequisites, such as the requirements for the feedback loop and a communication channel between the system and the framework. This contribution addresses **RQ3** and also **RQ4**, as the framework is an important prerequisite for executing the experiments;

- Experiments with a system whose requirements are based on analyses of the failure of a real system from a well-known case study in Software Engineering. Such experiments produced a series of models as the result of the application of the approach and simulations of real-world failures that allowed us to verify the response of the framework to a few undesired situations. This addresses **RQ4**.

Moreover, a fundamental difference from our approach and the state-of-the-art in goal-based adaptive systems design (presented later, in Section 2.2) is the fact that goals are not necessarily treated as invariants that must always be achieved. Instead, we accept the fact that the system may fail in achieving any of its initial requirements and, by considering feedback loops as first class citizens in the language, provide a way of specifying the level of criticality of each goal as constraints on their success/failure and assigning adaptation actions to be carried out when the system does not fulfill these constraints.

1.5 Structure of the thesis

To present the approach summarized in the previous section, the remainder of the thesis is structured as follows:

- Chapter 2 summarizes the state-of-the-art related to our work. First, the baseline for our proposal, including Goal-Oriented Software Engineering (GORE), Feedback

Control Theory and Qualitative Reasoning is presented. Then, other approaches for the design of adaptive systems are described in separate categories, for an easier comparison of related work;

- Chapter 3 characterizes the new requirements for system adaptation: Awareness Requirements and Evolution Requirements. The former specifies the requirements for the monitoring component of the feedback loop, whereas the latter prescribes requirements for the adaptation component;
- Chapter 4 presents a qualitative approach for adaptation through reconfiguration of system parameters that affect specific indicators of requirements convergence. Adaptation is guided by indicator/parameter differential relations which can be specified in different levels of precision, depending on available information;
- Chapter 5 proposes a systematic process for the design of adaptive systems based on the new modeling elements introduced in the previous chapters, including System Identification, the elicitation of Evolution Requirements and the adaptation strategies they compose;
- Chapter 6 describes in detail the run-time adaptation framework that acts as the controller in a feedback loop, reasoning over the requirements specification and the system's log and sending instructions on how to adapt;
- Chapter 7 reports the results of initial experimental evaluation conducted to validate our proposals, which consisted on modeling an Adaptive Computer-aided Ambulance Dispatch using our systematic process and simulating its failure to test the response of the run-time framework;
- Chapter 8 concludes the thesis with a summary of our contributions, its advantages and limitations, and the possibilities of future work that were opened by this research.

1.6 Published papers

We list here published work related to this thesis, split into *refereed* and *unrefereed*, and ordered by date of publication:

1.6.1 Refereed

- Souza, Vítor E. S. and Mylopoulos, John. Monitoring and Diagnosing Malicious Attacks with Autonomic Software. In Laender, Alberto; Castano, Silvana; Dayal,

Umeshwar; Casati, Fabio, and de Oliveira, José, editors, *Conceptual Modeling - ER 2009*, volume 5829 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009;

- Ali, Raian; Chopra, Amit K.; Dalpiaz, Fabiano; Giorgini, Paolo; Mylopoulos, John, and Souza, Vítor E. S. The Evolution of Tropos: Contexts, Commitments and Adaptivity. In *Proc. of the 4th International i* Workshop*, pages 15–19, 2010a;

- Souza, Vítor E. S.; Lapouchnian, Alexei; Robinson, William N., and Mylopoulos, John. Awareness Requirements for Adaptive Systems. In *Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 60–69. ACM, 2011b;

- Souza, Vítor E. S.; Lapouchnian, Alexei, and Mylopoulos, John. System Identification for Adaptive Software Systems: a Requirements Engineering Perspective. In Jeusfeld, Manfred; Delcambre, Lois, and Ling, Tok-Wang, editors, *Conceptual Modeling – ER 2011*, volume 6998 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2011a;

- Souza, Vítor E. S. and Mylopoulos, John. From Awareness Requirements to Adaptive Systems: a Control-Theoretic Approach. In *Proc. of the 2nd International Workshop on Requirements@Run.Time*, pages 9–15. IEEE, 2011;

- Ali, Raian; Dalpiaz, Fabiano; Giorgini, Paolo, and Souza, Vítor E. S. Requirements Evolution: From Assumptions to Reality. In Halpin, Terry; Nurcan, Selmin; Krogstie, John; Soffer, Pnina; Proper, Erik; Schmidt, Rainer, and Bider, Ilia, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 81 of *Lecture Notes in Business Information Processing*, pages 372–382. Springer, 2011a;

- Souza, Vítor E. S.; Mazón, Jose-Norberto; Garrigós, Irene; Trujillo, Juan, and Mylopoulos, John. Monitoring Strategic Goals in Data Warehouses with Awareness Requirements. In *Proc. of the 2012 ACM Symposium on Applied Computing*. ACM, 2012f;

- Souza, Vítor E. S.; Lapouchnian, Alexei, and Mylopoulos, John. (Requirement) Evolution Requirements for Adaptive Systems. In *Proc. of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (to appear)*, 2012d.

- Souza, Vítor E. S.; Lapouchnian, Alexei, and Mylopoulos, John. Requirements-driven Qualitative Adaptation. In *Proc. of the 20th International Conference on Cooperative Information Systems (to appear)*. Springer, 2012b.

1.6.2 Unrefereed

- Souza, Vítor E. S. An Experiment on the Development of an Adaptive System based on the LAS-CAD. Technical report, University of Trento (available at: `http://disi.unitn.it/~vitorsouza/a-cad/`), 2012;

- Souza, Vítor E. S.; Lapouchnian, Alexei; Robinson, William N., and Mylopoulos, John. Awareness Requirements for Adaptive Systems. In de Lemos, Rogério; Giese, Holger; Müller, Hausi A., and Shaw, Mary, editors, *Software Engineering for Self-Adaptive Systems 2 (to appear)*. Springer, 2012e;

  – Extended version of [Souza et al., 2011b];

- Souza, Vítor E. S.; Lapouchnian, Alexei, and Mylopoulos, John. System Identification for Requirements-Driven Qualitative Runtime Adaptation. *LNCS State-of-the-Art Survey Volume on Models@run.time (to appear)*, 2012c;

  – Extended version of [Souza et al., 2011a];

- Souza, Vítor E. S.; Lapouchnian, Alexei; Angelopoulos, Konstantinos, and Mylopoulos, John. Requirements-Driven Software Requirements Evolution. *Computer Science - Research and Development (to appear)*, 2012a;

  – Extended version of [Souza et al., 2012d].

Chapter 2

State of the art

A state-of-the-art calculation requires 100 hours of CPU time on the state-of-the-art computer, independent of the decade.

Edward Teller

Before we present our approach, we summarize the state-of-the-art in our chosen area of research. This chapter is divided in two main parts: first, Section 2.1 introduces background research used in subsequent chapters of this thesis, showing how this baseline was used to develop the initial models for the Meeting Scheduler, our running example. Then, Section 2.2 presents related work, i.e., other approaches that can be used for the design of adaptive systems.

2.1 Baseline

As mentioned in Chapter 1, the objective of this thesis is to define a requirements-based process for the design of adaptive systems centered on a feedback loop architecture. In the foundations of our approach there are *Goal-Oriented Requirements Engineering* (GORE) and *Feedback Control Theory*. Moreover, given our requirements perspective, our work also applies concepts of *Qualitative Reasoning* when modeling the requirements for adaptation. The next sections introduce the techniques and concepts that form the baseline of our proposal, with the aid of the running example of this thesis: the Meeting Scheduler.

2.1.1 Goal-Oriented Requirements Engineering

In the 1970s, Ross and Schoman Jr. [1977] proposed the *Structured Analysis and Design Technique* (SADT), recognizing the existence of the "requirements problem". According to Mylopoulos et al. [1999], this and other proposals to tackle the requirements problem

instituted the field of *Requirements Engineering* (RE). After roughly two decades of research, some people in this field started to recognize limitations of RE practices of the time, which lacked, for instance, support for reasoning about the composite system made of the software and its environment, support for understanding requirements in terms of their rationale, constructive methods for building correct models for complex systems and support for representation, comparison and exploration of alternatives [van Lamsweerde and Letier, 2002].

As a response, and inspired by well-established Artificial Intelligence techniques for problem solving, knowledge representation and knowledge acquisition [van Lamsweerde et al., 1991; Mylopoulos et al., 1992], new approaches were proposed around the concept of *goals*. A goal is a declarative statement of intent to be achieved by the system under consideration, formulated in terms of prescriptive assertions, covering different types of concerns — functional (representing services) or non-functional (representing qualities) — and different levels of abstraction — strategic (e.g., "optimize the use of resources") or technical/tactical (e.g., "send e-mail notifications to meeting participants"). Goals can refer to the current system in operation or to a system-to-be under development. They can also help indicate what parts of a system are or should be automated when responsibility for their satisfaction is assigned to a software agent (creating a requirement for automation), a human agent (a requirement for manual performance) or the environment itself (a domain assumption) [van Lamsweerde, 2001; van Lamsweerde and Letier, 2002].

Goals provide several advantages when compared to previously used concepts, for instance: precise criteria for sufficient completeness of a requirements specification (with respect to stakeholder goals); the rationale of each single requirement, thus justifying their pertinence; increased readability when structuring complex requirements documents; assistance in exploring alternative system proposals and in conflict detection and resolution; etc. [van Lamsweerde, 2001]. Furthermore, they are characteristically more stable than the processes, organizational structures and operations of a system which continuously evolve (although goals may also evolve when needed) [Antón, 1996]

Goal-oriented requirements elaboration processes end where most traditional specification techniques would start [van Lamsweerde and Letier, 2002], thus driving the identification of requirements [van Lamsweerde, 2001]. Goal-oriented analysis amounts to an intertwined execution of analyses of non-functional requirements, of functional requirements, and conflict analysis, and can be declared complete when all relevant goals have been operationalized by the new system [Mylopoulos et al., 1999].

Many different approaches that follow the principles of GORE have been proposed. In the early days, Kaindl [1997] proposed *Requirements Engineering Through Hypertext* (RETH), a pragmatic approach that combines Object-Oriented Analysis with require-

ments. The NFR Framework [Mylopoulos et al., 1992] promotes the analysis of different solutions according to non-functional requirements modeled as softgoals. KAOS [van Lamsweerde et al., 1991; Dardenne et al., 1993], which stands for *Knowledge Acquisition in autOmated Specification* (or, alternatively, *Keep All Objectives Satisfied*), provides a conceptual model, an associated language and a set of strategies for requirements acquisition based on goals. Antón's [1996] *Goal-Based Requirements Analysis Method* (GBRAM) offers techniques for analyzing, elaborating and refining goals.

$i^\star$ [Yu and Mylopoulos, 1994] (more recently, [Yu, 2009; Yu et al., 2011]), which stands for *distributed intentionality*, introduced aspects of social modeling and reasoning into RE, by putting social concepts into the core of the daily activity of system analysts and designers. Later, the Tropos methodology [Castro et al., 2002; Giunchiglia et al., 2003; Bresciani et al., 2004] adopted $i^\star$ and took its concepts beyond early requirement stages and throughout the software development process. Also founded on $i^\star$, the *Goal-oriented Requirements Language* (GRL)[1] is part of an ITU-T Recommendation[2] for systems development called *User Requirements Notation* (URN), supporting goal modeling and reasoning. According to Yu [2009], URN brings together $i^\star$'s social and intentional modeling with the scenario-oriented approach of Use Case Maps (UCM).

A report by Lapouchnian [2005] provides an overview of GORE and further information on some of the approaches cited above. Another overview of this field can be seen in [van Lamsweerde, 2001].

2.1.2 Illustrating GORE concepts: the Meeting Scheduler example

To illustrate some of the above concepts, we will consider the case of large companies that have many employees who need to conduct meetings on their day-to-day business, using one of several meeting rooms and other resources such as projectors, teleconference equipment, etc. Although not a real case study, the meeting scheduler example was based on the author's experience during a consultancy project on business process modeling within one of the business units of Petrobras,[3] Brazil's largest oil/energy company. The project involved many meetings for business process analysis with other employees of the company and the way meeting scheduling is described in this thesis mirrors what the author had to do to organize meetings in that setting.

Goals could be used to model the meeting scheduling system-as-is, using, for instance, $i^\star$'s Strategic Dependency diagrams as depicted in Figure 2.1. The diagram shows the current social setting that has been established in order to accomplish the objective of

[1] `http://www.cs.toronto.edu/km/GRL/`.

[2] `http://www.itu.int/rec/T-REC-Z.151/en`.

[3] `http://www.petrobras.com/en/about-us/`.

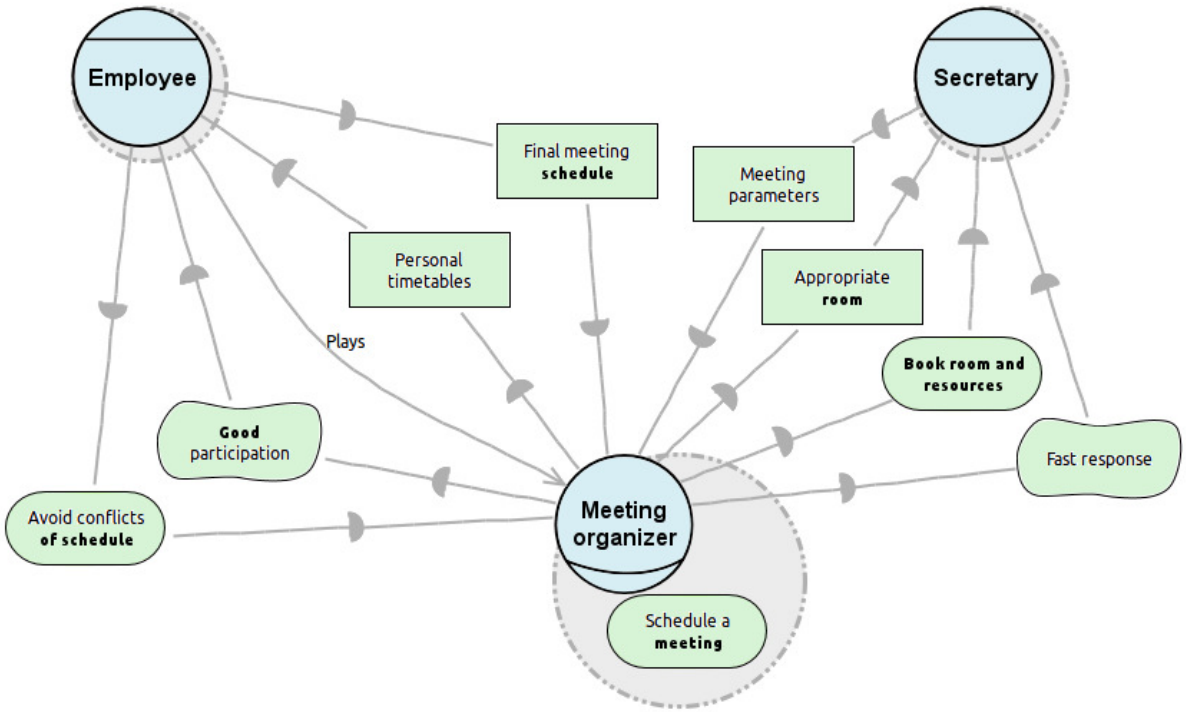


Figure 2.1: Strategic Dependency model of the social environment around meeting scheduling.

scheduling meetings.

In this setting, an employee plays the role of the meeting scheduler when a meeting is needed. To accomplish her goal of scheduling the meeting, she depends on the employees participating in the meeting to send her their personal timetables in order to decide the best time for the meeting. Then, she has to obtain an appropriate room (preferably fast) from a secretary, providing the meeting parameters (number of people, desired date/time, necessary equipment, etc.) and depending on the secretary to actually book the room and other needed resources. Once the schedule is final, employees depend on the meeting organizer to inform them about the meeting and to avoid conflicts of schedule (between the scheduled meeting and their personal appointments). Finally, the meeting organizer expects good participation (attendance) from the invited employees.

Imagine, now, that the aforementioned actors plus the company's board of directors are not satisfied with this system and would like to improve it by automating scheduling by means of a software. Many non-GORE software development processes would start by collecting the requirements for this software using, for instance, scenarios, use cases or stories. Using a goal-oriented perspective, however, we take a step back and instead of modeling the *requirements for a solution* directly, we analyze the *problems* stakeholders find in the current system, identifying the *strategic goals* of the stakeholders. Strategic

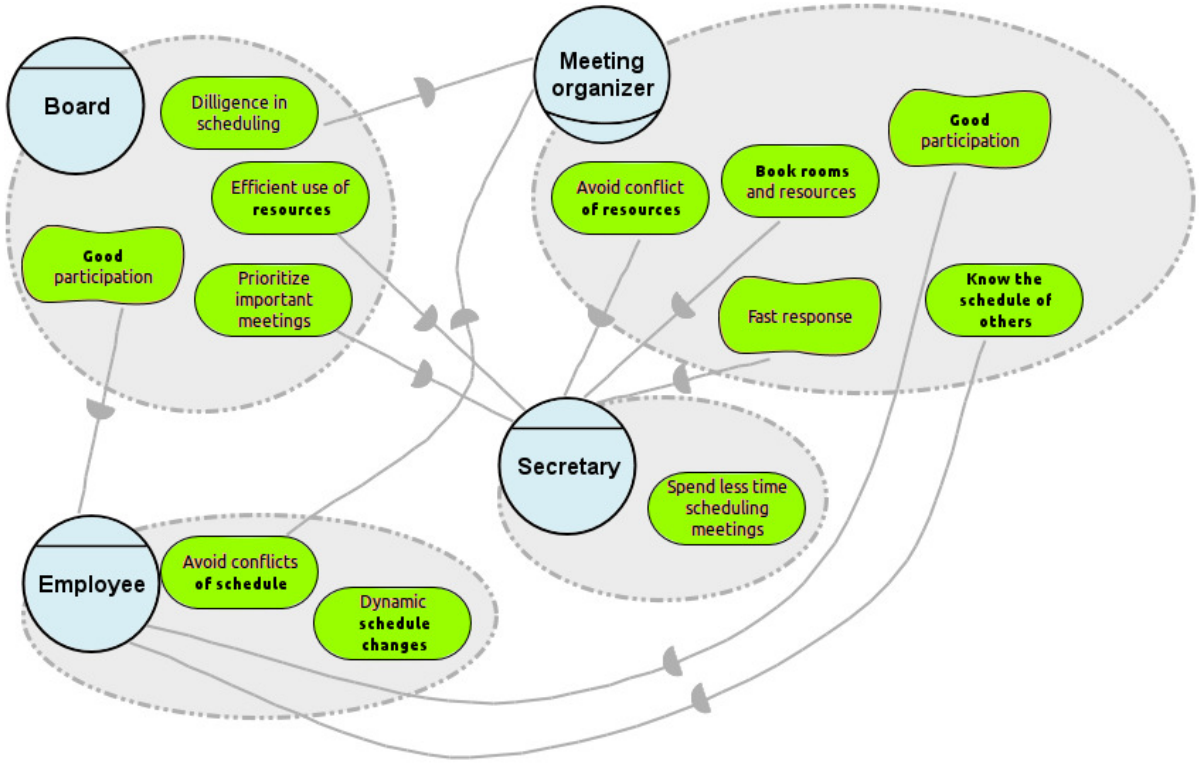


Figure 2.2: An $i^\star$-like diagram depicting the strategic goals of the stakeholders.

goals represent the rationale for the development of a system, answering the question "why should we develop this or that solution?"

Figure 2.2 shows, again using $i^\star$, the strategic goals of each stakeholder. The goals have been colored differently to denote their strategic level and are summarized next:

- The **board** of directors has identified that, in the current system, secretaries are overloaded and, thus, usually provide rooms for meetings regardless if they are unnecessarily larger, which wastes company resources. Furthermore, the current system doesn't allow them to evaluate which meetings have greater priority when booking rooms and resources. For the board is also important to enforce diligence in scheduling (e.g., if a meeting has been canceled, its booking should also be) and promote good participation, but currently there is no way to guarantee either of them;

- **Meeting organizers** depend on secretaries to book rooms and resources as fast as possible and avoiding conflicts but, again due to their overload, these goals are not being satisfied. Also, in order to choose a good date and time for meetings, organizers depend on knowing the timetables of other employees, but currently this is a long and tedious process. Like the board of directors, good participation is also

important for meeting organizers;

- **Employees**, in their turn, would like meeting organizers to avoid to schedule meetings when they are busy and, therefore, cannot participate. Reconciling all participant's schedules to guarantee the satisfaction of this goal is very difficult and currently not always guaranteed. Employees also mentioned that they would like to have the freedom to change their own schedules even if this creates conflicts with meetings that are already booked, having the meeting schedules automatically adapted to the changes. This goal is not even considered in the current system;
- Finally, all the **secretaries** want is to spend less time scheduling rooms for meetings, because it is a tedious and time-consuming task, which gets in the way of their many other duties. This is obviously not considered in the current situation, as they are currently responsible for room and resource booking.

Modeling the strategic goals of the stakeholders can also help in deciding the best alternative to solve the problems at hand. Take, for instance, the model of Figure 2.3: the diagram shows different possible solutions to satisfy the secretaries' goal *Spend less time scheduling meetings*. Using means-end relationships, the diagram states that this goal can be satisfied either by (a) asking meeting organizers to help and not place unnecessary load on the secretaries; (b) train secretaries to perform better; (c) replace the secretaries with more capable ones; (d) hire more secretaries to help out; or (e) develop a scheduling software that automates many of their tasks. Using qualitative contribution links to softgoals that represent quality criteria (which should also be elicited from the stakeholders) helps identify which would be the best solution to be implemented. In real projects, however, a more thorough feasibility study and cost-benefit analysis is advised.

Once the solution to be implemented has been chosen, the social setting for the system-to-be can be represented in a new Strategic Dependency diagram, as shown in Figure 2.4. The diagram shows that stakeholders now depend on this new software system to satisfy all of their strategic goals, thus setting a criteria for the completeness of the requirements for the new system: the requirements have to satisfy all of the stakeholder goals.

The software-intensive meeting scheduler system has as main goal *Schedule meeting*, which is represented in a different color to indicate a different type of concern: while strategic goals referred to the problems, tactical/technical goals detail the solution, i.e., the functions/services to be provided by the software system. Although not illustrated here, we can continue to harness the benefits of social modeling at the tactical level, by decomposing the scheduler system's top goal until the level of tasks that can be operationalized by the software (e.g., using $i^\star$ Strategic Rationale diagrams). By performing this decomposition, the socio-technical aspects of the system can become more evident

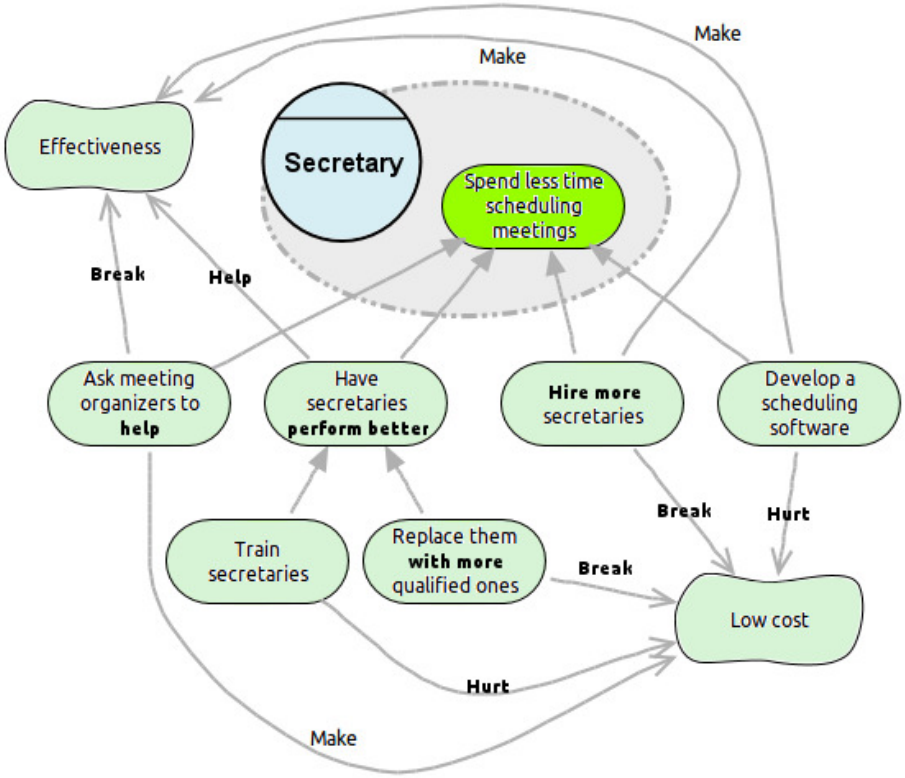


Figure 2.3: Evaluating alternative solutions for the problem identified by the secretaries.

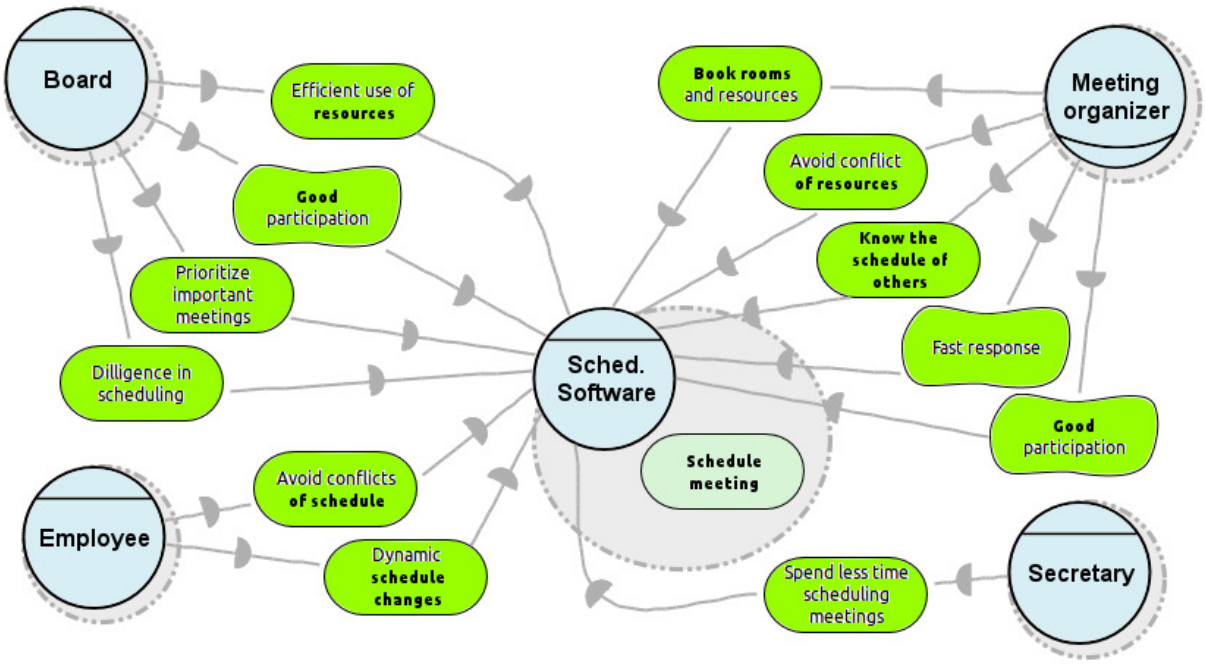


Figure 2.4: Strategic Dependencies for the meeting scheduler system-to-be.

through dependency links from the software to one or more human actors (e.g., "secretaries need to confirm that meetings really happened", "organizers have to provide the system with the list of participants to invite", etc.). Furthermore, how the system satisfies the stakeholder goals can become more evident by modeling contribution links from tactical goals to strategic goals.

The process that this sub-section illustrated, sometimes identified as *Early Requirements*, is one among many possible ways of using GORE concepts to analyze a particular problem with the purpose of designing a specification for a software-intensive system that satisfies the stakeholder goals. Our proposals in this thesis, however, do not prescribe any particular GORE methodology for early requirements, but does expect a goal-oriented perspective in *Late Requirements*, i.e., that the specification for the solution be a goal model. The next section discusses this further.

2.1.3 GORE-based specifications

After analyzing the stakeholder goals in early requirements phase and choosing a solution, late requirements is then concerned with determining a specification S that, together with domain knowledge K, satisfies the requirements R. Or, as Zave and Jackson [1997] put it: $S, K \vdash R$. Our models, however, are based on the revised core ontology for RE proposed by Jureta et al. [2008], which defines as primitive concepts *goals*, *tasks/plans*, *softgoals*, *quality constraints* (QCs) and *domain assumptions* (DAs). Figure 2.5 shows an initial specification for the meeting scheduler system, using these primitive concepts.

Goals, modeled by ovals, represent states of affairs that the actor wants to achieve, for instance, *Schedule meeting* (or, put another way, "the state in which the meeting is scheduled"). In our late requirements models, the actor is implicitly the system-to-be — note that in Figure 2.4, the goal *Schedule meeting* belongs to the *Scheduling software*. This does not necessarily means, however, that the goal is to be achieved autonomously, without any humans in the loop. As mentioned in Chapter 1, socio-technical systems include both software/hardware and human/organizational elements.

Tasks, modeled by hexagons, represent a sequence of actions to be conducted by the actor (again, in a socio-technical system this could involve humans in the loop), usually with the purpose of achieving some goal. In Figure 2.5, *Characterize meeting* represents the series of steps taken by the meeting organizer using the scheduling software in order to specify a meeting's preferred date/time, list of participants, needed equipment and detailed description. Tasks can be directly related to procedures/methods implemented in code.

Domain assumptions, modeled by (square-cornered) rectangles, represent states of affairs that are assumed by the actor to always be true. For example, it is assumed

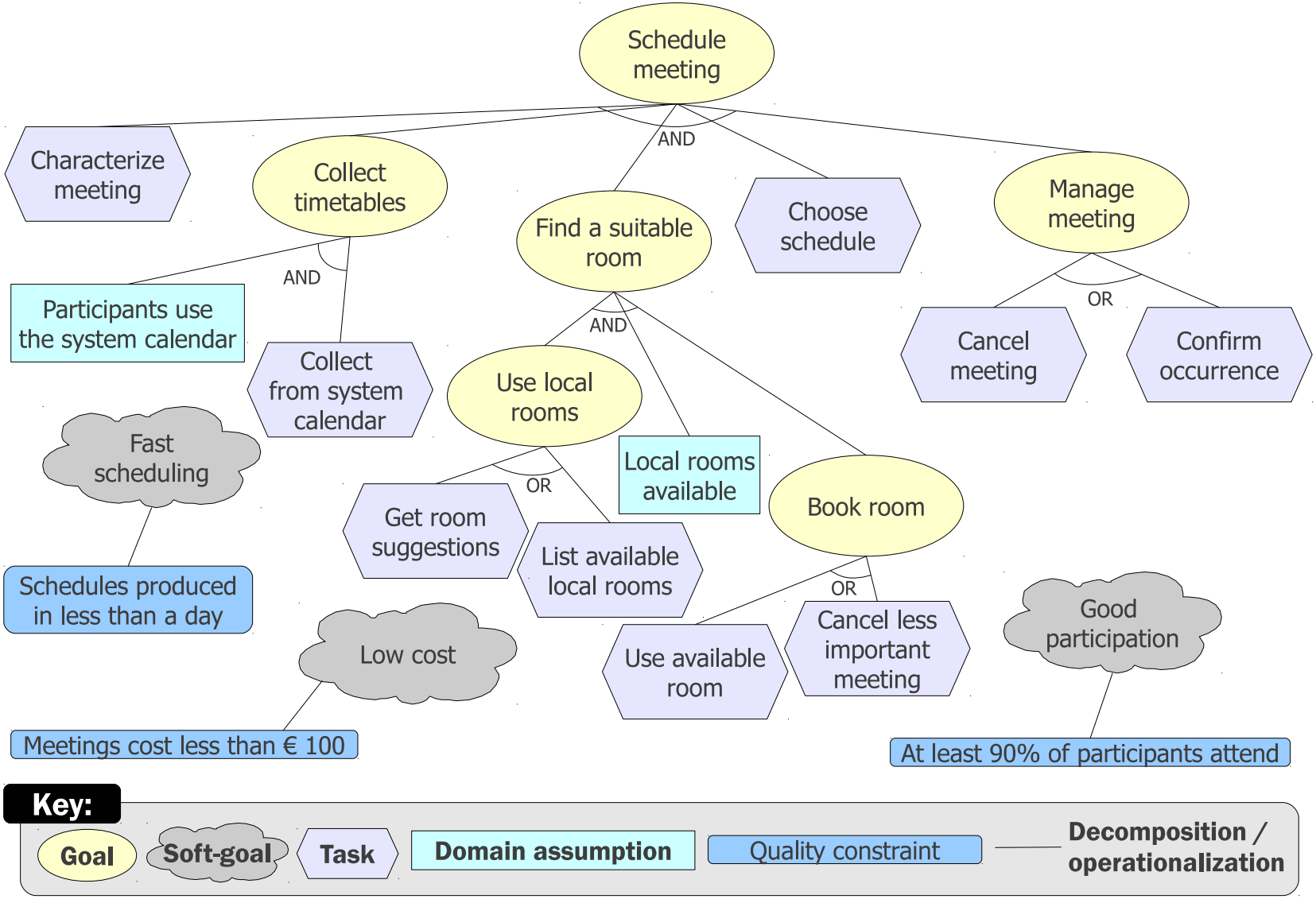


Figure 2.5: Initial specification for the meeting scheduler.

that *Participants use the system calendar*, and, therefore, the system (which is the actor in this case) relies on this fact being true in order to satisfy the requirements. Domain assumptions relate to Zave and Jackson's [1997] domain knowledge.

Softgoals, modeled by clouds, are like (hard) goals, but have no clear-cut criteria for satisfaction. For instance, *Fast scheduling* indicates that an important quality expected from this system is not to take too long a time to schedule a meeting, but the softgoal itself does not indicate how much time is too long. In contrast, hard goals like *Schedule meeting* do provide a clear criteria in themselves: either the meeting was scheduled, or it was not. Softgoals usually represent (non-functional) concerns that cut across many other elements of the specification. As such, they are represented as top-level elements, by themselves.

Finally, since for our purposes it is important to determine if requirements were satisfied or not, ***quality constraints*** (van Lamsweerde [2009] calls them *Fit Criteria*), modeled by round-cornered rectangles, provide a clear-cut criteria for the satisfaction of softgoals. Taking the *Fast scheduling* example again, QC *Schedules produced in less than a day* provide the precise criteria for its satisfaction: schedules produced in less than 24 hours are considered fast, whereas schedules that take more than that time to be produced are not.

In the model of Figure 2.5, the top goal is then further refined using a relation that has a single syntax (a solid line) but two possible semantics (decomposition or operationalization), as following:

- When connecting two elements of the same type, the association represents a ***decomposition***, i.e., a part-whole (AND-decompositions) or specialization (OR-decompositions) relation between the parent element and its children (as a convention, parents are positioned above their children). For example, *Collect timetables*, *Find a suitable room* and *Manage meeting* are all children of *Schedule meeting*, i.e., they represent partial states of affairs, which are parts of a whole, the *Schedule meeting* goal;

- When connecting goals to tasks or domain assumptions, or softgoals to quality constraints, the association represents an ***operationalization***, i.e., a means by which the (soft)goal can be satisfied. The concept of operationalization links is not new, and was introduced by van Lamsweerde and Willemet [1998] for the KAOS language. For instance, for the meeting scheduler, other than satisfying the aforementioned sub-goals of *Schedule meeting*, to achieve the top goal of Figure 2.5 one has also to *Characterize meeting* and *Choose schedule*. To satisfy *Collect timetables*, given that *Participants use the system calendar* is true, it is enough to *Collect [the timeta-*

bles] from system calendar. As mentioned before, QCs provide clear-cut criteria for softgoals and each softgoal in our example is operationalized by one QC.

Refinement links — which is how we refer to both decomposition and operationalization indistinguishably — can be of two types, *AND* or *OR*, with obvious semantics: an AND-refinement means that in order to satisfy the parent (soft)goal, all of its children must be satisfied, while for an OR-refinement, only one child needs to be attained. As for the other elements, tasks are satisfied if they are executed successfully, domain assumptions are satisfied if they hold (the affirmation is true) while the user is pursuing its parent goal and the satisfaction of quality constraints is domain-dependent (the ones in Figure 2.5, for instance, should be checked for each meeting that is scheduled).

We henceforth use this syntax for our goal models for two main reasons. First, it highlights the fact that our proposals can be applied to any GORE methodology that contains the primitive elements described above, not restricting itself to one specific approach. Second, using a single syntax for decomposition and operationalization makes our models much more concise. To represent some of our refinements in $i^\star$, for instance, one would have to use a combination of means-end links and task decompositions that would make the model unnecessarily bigger.

Back to Zave and Jackson's [1997] statement — $S, K \vdash R$ — and considering that our models are composed of sets G_H (hard goals), G_S (softgoals), T (tasks), D (domain assumptions) and Q (quality constraints), we can relate to Zave and Jackson's proposal the following way:

- $R = G_H \cup G_S$: the requirements for this system is that it satisfies all goals. Here, in order to avoid having vague requirements, we consider that to satisfy the softgoals one should use the criteria established by the quality constraints associated with them;
- $S = T \cup Q$: the specification for this system is the set of tasks and quality constraints that operationalize the goals. Tasks specify exactly what is to be implemented, whereas quality constraints affect the way these tasks are implemented;
- $K = D$: the domain knowledge consists of the domain assumptions made by the stakeholders. The terminology change (knowledge $\rightarrow$ assumptions) show that DAs are intentional, i.e., they represent the fact that the stakeholders agree on what should be assumed to be true, as opposed to inferences made by a developer's analysis of the domain.

Furthermore, we believe that the specification plus the domain knowledge not only *infer* the requirements ($\vdash$, syntactic consequence), but they actually *entail* them ($\models$,

semantic consequence). Therefore, the final equation for GORE-based requirement specifications is:

$$T \cup Q, D \models G_H \cup G_S \tag{2.1}$$

2.1.4 Variability in goal models

In Section 2.1.2, we have shown how early requirements GORE approaches allow analysts to model different solutions for the same problem in order to decide the best one to develop. In Section 2.1.3, a specification for the chosen solution was illustrated in Figure 2.5 (p. 33), using the meeting scheduler example.

Note, however, that this solution lacks an important characteristic for adaptive systems: different ways of accomplishing the same goals. Granted, to *Use local rooms* one can *Get room suggestions* or *List available rooms*. Moreover, to *Book room* one can *Use available room* or *Cancel less important meeting*. However, timetables should always be collected using the system calendar, only local rooms should be used and the schedule has to be produced manually. As briefly explained in Section 1.4.2 (p. 17), adaptation can be done by switching the system's configuration in order to use a different means to satisfy its goals, so this sort of redundancy — the representation of *alternatives* or *variability* — is very important for adaptive systems.

Figure 2.6 shows a new goal model for the meeting scheduler, with added variability. The sub-trees of goals *Collect timetables*, *Find a suitable room* and *Choose schedule* were changed to address the issues that were mentioned in the previous paragraph.

This topic of research has been quite explored by the literature. According to van Lamsweerde [2009], variability in the design of software-intensive systems can come from many different sources, such as goal refinements (decomposition/operationalization), different countermeasures for risks, different resolutions for conflicts, and different actors to which goals/tasks can be assigned. These situations lead to design decisions, which in turn lead to different system proposals and different software architectures.

In our case, the model of Figure 2.6 contains different ways of achieving the system's goals not to allow the design-time choice of the best solution to be implemented, but to allow the run-time reconfiguration of the system for adaptation. Therefore, the set of tasks T in the specification is not the minimal set that can satisfy the top goal by applying Boolean upwards propagation of refinements, but is the whole set of tasks in the model. The same goes for quality constraints. If any task is not implemented and at run-time the system decides to reconfigure and use that particular task as a solution, the system will have to wait for developers to implement it before proceeding, which in many cases is not an appropriate response.

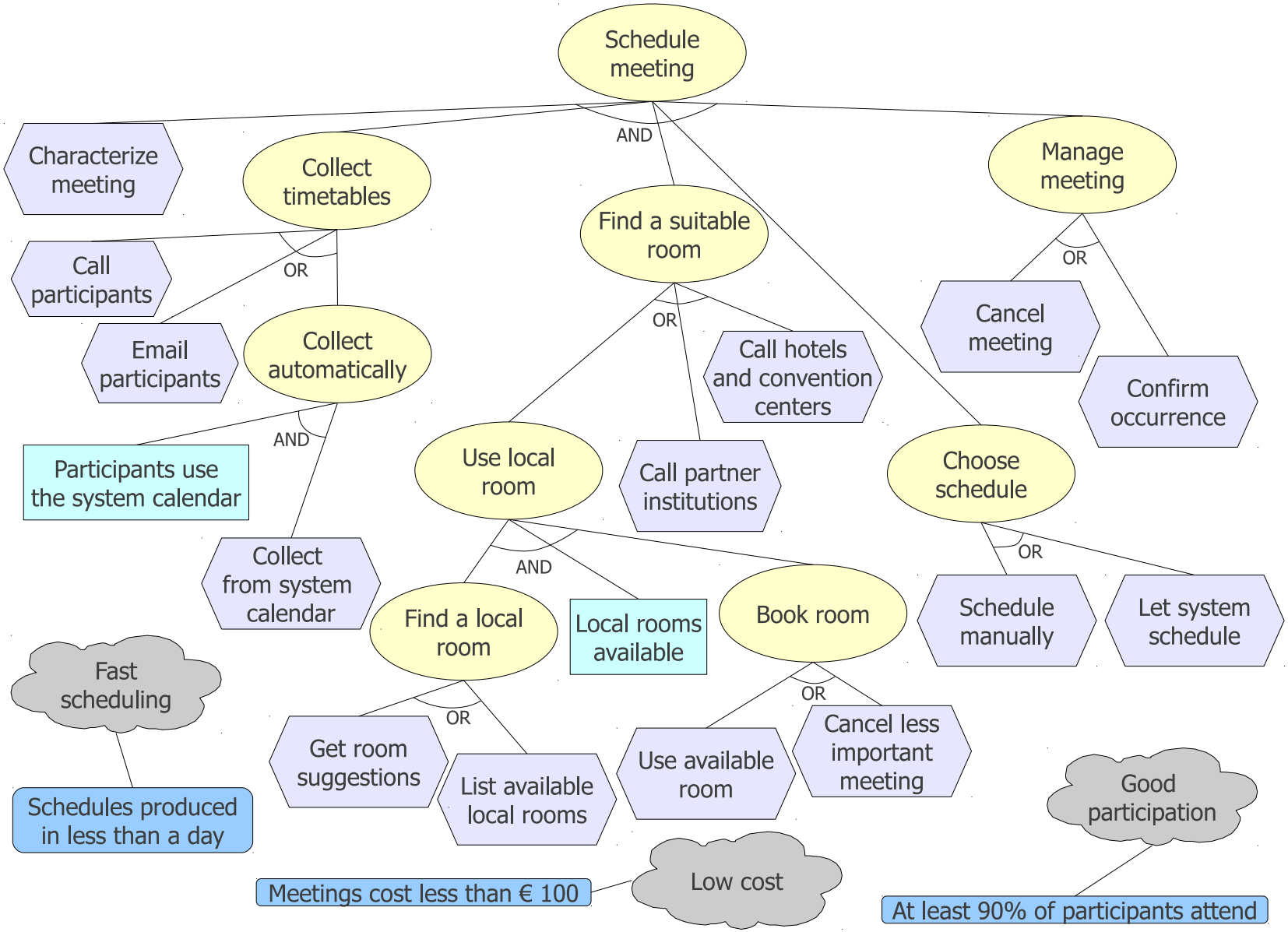


Figure 2.6: Final specification for the meeting scheduler, with added variability.

Other researchers have used variability for other purposes. To cite a couple of examples, Gonzalez-Baixauli et al. [2004] represent the variants of a product family using goal models (amenable to formal analysis) and, based on the NFR Framework, use softgoals as criteria to select a *satisficing* solution. Also in the field of Software Product Lines (SPL), Semmak et al. [2008] extend the KAOS approach with variability to promote reuse during requirements engineering.

Research on requirements for context-aware systems also addresses the issue of variability (which is expected, since context-aware systems are like adaptive systems, but with an open loop instead of a feedback loop, cf. Section 1.2.2, p. 7). Hong et al. [2005] touches the issue of variability in the research on Human Computer Interaction (HCI) to address the requirements of ubiquitous applications in three different categories of contexts: computing, user and physical. Salifu et al. [2007] uses concepts from SPL to propose an approach to identify, represent, analyze and reason about variants in the descriptions of product families using problem frames.

Specifically for GORE, Liaskos et al. [2006] propose an approach to goal decomposition to support requirements elicitation for highly customizable (i.e., variability-intensive) software, characterizing OR-decomposition of goals semantically and identifying variability by analyzing stakeholder speech using Linguistic tools. Liaskos et al.'s work only captures intentional variability, and was therefore extended in [Lapouchnian and Mylopoulos, 2009] to capture also domain variability. In this extension, the authors propose a formal framework that defines contextual tags specified by Boolean statements based on domain properties/assumptions and allows these tags to be organized in hierarchies and assigned to elements of the goal model, thus determining when each element is active or not, depending on the context. Ali et al. [2010b] has a similar proposal, in which the Tropos methodology [Bresciani et al., 2004] is extended with a set of modeling constructs to analyze, elicit and model relevant context information, plus reasoning techniques for run-time derivation of goal model variants that reflect the current context / user priorities and design-time derivation of specifications that cover all considered contexts.

As with the basic GORE methodology to use, our approach does not prescribe any specific technique for elicitation, analysis and modeling of variability, be it domain-related or intentional, and the analyst is free to use the one that fits her best. We do recommend that the specification for the adaptive system-to-be is done with the sort of redundancy illustrated in this sub-section in order to allow for the usage of reconfiguration as an adaptation approach. In chapters 3 and 4, when we revisit the Meeting Scheduler example to illustrate our modeling constructs, we will come back to Figure 2.6 as the final specification for the non-adaptive version of this system.

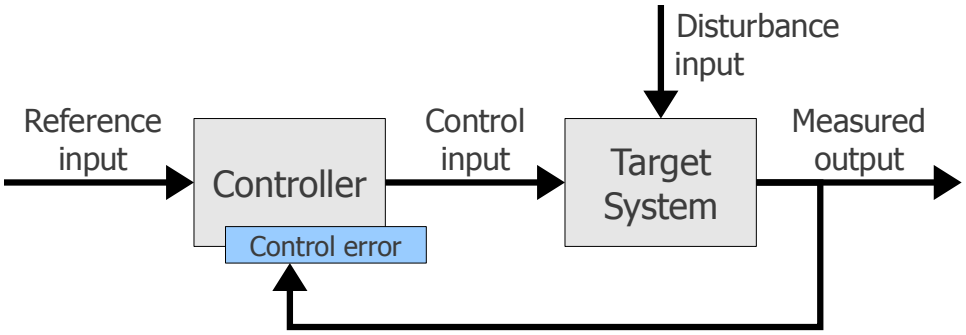


Figure 2.7: Simplified block diagram of a control system based on [Hellerstein et al., 2004].

2.1.5 Feedback Control Theory[4]

In our research, we assume the architecture for the design of an adaptive system uses one or more feedback loops to implement adaptivity. In other words, we see adaptive systems as feedback control systems, borrowing concepts from the field of *Control Theory* [Doyle et al., 1992].

Figure 2.7 shows a simplified view of a control system, adapted from [Hellerstein et al., 2004]. In this kind of system, the *reference input* is "the desired value of the measured outputs",[5] while the *measured output* is "a measurable characteristic of the target system". For instance, consider a (simplified view of a) car's cruise control mechanism, which is a classic example of a control system. Its purpose is to maintain the car at some constant speed S_I. In this example, S_I is the *reference input*, whereas the actual speed of the car S_O, which can be read from the car's speedometer, is the *measured output*.

Given this information, the *controller* "computes values of the control input based on current and past values of control error". The *control error* is "the difference between the reference input and the measured output", while the *control input* is "a parameter that affects the behavior of the target system and can be adjusted dynamically". Back to the example, the *control error* E can be calculated as $E = S_I - S_O$, leading to a straightforward definition for the *control input*: if $E > 0$, the *controller* (the cruise control system) should inject more fuel in the engine to speed up the vehicle (the *target system*). Analogously, if $E < 0$, less fuel should be injected. The idea is to keep S_O as close as possible to S_I at all times.

Finally, the *disturbance input* "are factors that affect the measured output but for which there is no governing control input". In other words, these are taken from the

[4]Another field of study that deals with systems involved in a closed signal loop is Cybernetics. During work on this thesis, however, we have focused on Control Theory as baseline for our proposals.

[5]This and the following quotes were taken from Hellerstein et al. [2004], §1.1.

context in which the system executes. Neither the system nor the controller have any control over these values. For the cruise control system, the inclination of the road and the direction and strength of the wind are examples of *disturbance inputs*, as they can have an influence on the measured speed S_O.

Another classic example of a control system is a thermostat that regulates the temperature in a room. In this case, the *reference input* is the desired temperature, the *target system* is the heating/cooling device, the *measured output* is the actual temperature in the room and the *control input* is the amount of gas/electricity the *controller* will send to the heating/cooling device in order for it to increase or decrease the room temperature.

Simple control systems like these are said to be SISO, meaning *single input, single output* (e.g., desired and actual temperature, respectively). Such systems are usually handled by a PID controller, widely used in the process control industry [Hellerstein et al., 2004]. The PID controller consists of three components, or *modes of control*:

- **P**roportional: *proportional control* sets the value of the control input proportional to the current control error. For instance, if the room is 10° cooler than desired, heating power is set to $10 \times K$, if it is 5° cooler, set to $5 \times K$, and so forth. Therefore, as the error decreases, so does the corrective action. The constant K is called *proportional gain* and has to be tuned in order to avoid a controller that is too conservative (K is too low, so it takes too long to adapt) or too unstable (K is too high, so it overshoots the heater and the room gets too hot, then it overshoots the cooler and the room gets too cold, etc).

- **I**ntegral: *integral control* addresses a limitation of the proportional component: it cannot handle disturbances well. For instance, imagine the room is 5° cooler than desired and the heating is turned up with $power = 5K$. For some reason, however, the window is open and it turns out that the heating produced by the heater is exactly the opposite of the cooling produced by the open window, resulting in the proportional controller not being able to reduce the control error. The integral mode works by setting the control input proportional not to the current control error, but to the sum (i.e., the integral) of the past control errors, e.g., at time t_0, $sum = 0$; at t_1, $sum = 5$; at t_2, $sum = 10$, etc. Therefore, as the time passes, the integral control becomes "stronger" and increases the control input (e.g., increases the heating power).

- **D**erivative: *derivative control* is used to provide better performance. Depending on how it is tuned, a $P+I$ controller might take too long to reach stability either because the change is too conservative (long time to reduce the control error) or too aggressive (long time overshooting and oscillating around the desired value). To overcome this,

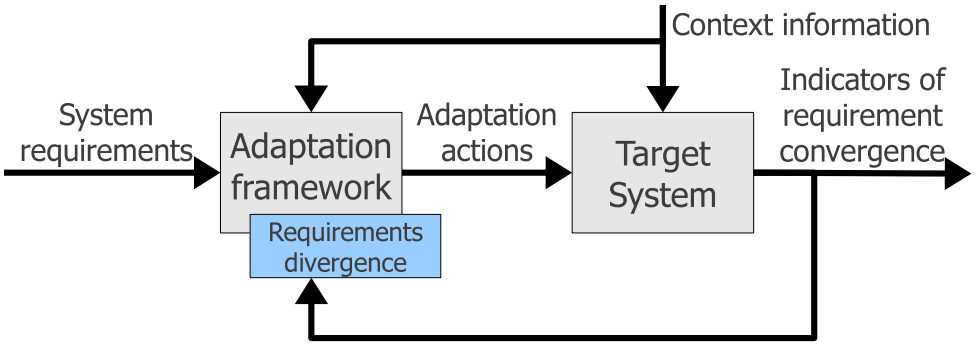


Figure 2.8: View of an adaptive system as a feedback control system.

the derivative mode looks at the rate of change (i.e., the derivative) to predict if the output is changing too slow or too fast, adjusting the control input accordingly. However, because this mode of control can be highly affected by monitoring noise, it is sometimes considered dangerous, making the $P+I$ combination more commonly used.

The PID controller works well, but more complex systems, such as information systems, usually have *multiple inputs, multiple outputs* (MIMO). State-of-the-art methods for modeling and controlling MIMO systems — such as state/output feedback and Linear Quadratic Regulator (see [Zhu et al., 2009], Section 3.4) — can be very complex and many software projects may not dispose of the necessary (human/time) resources to produce models with such degree of formality.

For these reasons, our proposals consider adaptive systems as simplified control systems, taking inspiration on the PID controller. This view is represented in Figure 2.8. The different inputs and outputs for feedback control systems translate to adaptive systems as follows:

- **Reference input** (requirements): in an adaptive system, the *reference input* consists of the *system requirements*. As discussed in previous sub-sections, we adopt GORE-based requirements specifications, e.g., the goal model depicted in Figure 2.6 (p. 37) for the Meeting Scheduler example. As will be shown in later chapters of this thesis, however, *system requirements* should include not only "vanilla" (i.e., not concerning adaptation) requirements such as the ones illustrated in Figure 2.6, but also adaptivity requirements.

- **Measured output** (indicators): if requirements are the *reference input*, the *measured output* should then consist of *indicators of requirements convergence*. In other words, we would like to measure, at runtime, if functional requirements are being

met (e.g., are users able to successfully *Find available rooms* for their meetings? What is the success rate for this goal?) and what are the degrees of satisfaction of non-functional requirements (e.g., what is the participation percentage for each scheduled meeting? Is it above 90%? What is the success rate for this quality constraint?). Such indicators are usually of Boolean nature (i.e., satisfied: true/false) and measures in other domains (e.g., the response time of a task or the success rate of a goal, both numeric) can be mapped to Boolean by a function that maps each value of the domain to satisfied/unsatisfied (in the case of numeric values, a threshold usually provides such mapping).

- **Control error** (divergence): given the above *reference input* and *measured output*, the *control error* consists of a set of requirements (be they goals, quality constraints or even domain assumptions) that were not satisfied either individually (i.e., during a single execution of the system) or in an aggregate way (average success rate). Negative answers to questions presented previously (Are user able to successfully use the system? Is participation above 90%? Are 95% of all meeting participants using the timetable database?) are examples of *requirements divergence*.

- **Control input** (adaptation): given the information on the *control error*, the *control input* consists, of course, of the adaptation actions, which might include reconciliation of system behavior and compensation to avoid inconsistent system states. Later in this thesis we present two approaches to adaptation: reconfiguration and evolution (cf. Section 1.2.4, p. 10).

- **Disturbance input** (context): the factors that can be measured but that neither the *target system* nor the *adaptivity framework* have any control over are called *context information*. Unlike the *disturbance input* in control systems, though, *context information* in adaptive systems could be provided as input not only to the *target system*, but also to the *adaptation framework*. The reason for this is that the controller itself can be context-sensitive, selecting appropriate *adaptation actions* depending on the context. However, this thesis does not address this issue with the necessary depth and regard the impact of contexts in our approach as future work.

Considering adaptive systems as feedback control systems comes from the realization that, at runtime, things might not go as planned. As discussed earlier, in sub-section 2.1.3 (p. 32), our specifications consist of the set of tasks T and the set of quality constraints Q that, when considered together with domain assumptions D, satisfy the system's goals and softgoals. Of course, this is an optimistic view of the world. The tasks that are part of a specification may actually not be carried out during any one execution, or may not have

the expected effects because of a fault. Also, quality constraints may not be satisfied. And domain assumptions may not hold in particular circumstances. For example, we assume all *Participants use the system calendar* in the Meeting Scheduler system, but is that really the case at runtime? This is useful information to monitor for when trying to satisfy the goal *Collect timetables*.

2.1.6 Requirements monitoring

The discussion in the previous sub-section highlights the fact that, in order to use system requirements as the reference input to a feedback loop implementing adaptivity, requirements have to be monitored for their satisfaction at runtime. Requirements monitoring research dates back to before there was this increased interest in adaptive / autonomic systems by RE researchers.

The seminal work of Fickas and Feather [1995] already recognized that many systems are deployed in environments that cannot be counted to remain static and, therefore, "requirements monitors [should] be installed to gather and analyze pertinent information about the system's run-time environment". Their approach relates the system requirements to assumptions made about the environment in which the system operates, generating monitors to detect when relevant changes to this environment take place. When a mismatch between the assumptions and the current environment are detected, remedial evolutions of the system's design are applied.

The work of Fickas and Feather was further explored in [Feather et al., 1998], in the context of the KAOS methodology. This approach uses the Formal Language for Expressing Assumptions (FLEA) to define, at design-time, events that represent violation of assumptions over the system requirements. Moreover, reconciliation tactics are identified and associated with each possible violation, in order to be applied at runtime when FLEA fires any of the defined violation events. This work combined, for the first time, goal-based requirements-time reasoning, event-based run-time monitoring and system self-adaptation tactics.

Central to this thesis is the research of prof. William N. Robinson, which proposed the *ReqMon* framework [Robinson, 2005, 2006], including a language for the definition of requirements monitors and a methodology for the identification of potential requirements obstacles and analysis of monitor feedback. Later on, this framework would be extended into *SerMon* to monitor service systems [Robinson and Purao, 2011] and eventually generalized into the *Event Engineering and Analysis Toolkit* (EEAT).[6]

EEAT provides a programming interface (API) that simplifies temporal event reasoning. It defines a language to specify goals and can be used to compile monitors from the

[6] `http://eeat.cis.gsu.edu:8080/`.

goal specification and evaluate goal fulfillment at runtime. Such monitors are specified using a variant of the Object Constraints Language (OCL), called OCL_{TM} — meaning OCL with Temporal Message logic [Robinson, 2008]. OCL_{TM} extends OCL 2.0[7] with:

- Flake's [2004] approach to messages: replaces the confusing `^message()`, `^^message()` syntax with `sentMessage/s`, `receivedMessage/s` attributes in class `OclAny`;
- Standard temporal operators: $\circ$ (`next`), $\bullet$ (`prior`), $\Diamond$ (`eventually`), $\blacklozenge$ (`previously`), $\Box$ (`always`), $\blacksquare$ (`constantly`), $\mathcal{W}$ (`always ...unless`), $\mathcal{U}$ (`always ...until`);
- The scopes defined by Dwyer et al. [1999]: `globally`, `before`, `after`, `between` and `after ...until`. Using the scope operators simplifies property specification;
- Patterns, also in Dwyer et al. [1999]: `universal`, `absence`, `existence`, `bounded existence`, `response`, `precedence`, `chained precedence` and `chained response`;
- Timeouts associated with scopes: e.g. `after(Q, P, '3h')` indicates that `P` should be satisfied within three hours of the satisfaction of `Q`.

Listing 2.1 shows an example of OCL_{TM} constraint on the Meeting Scheduler. The invariant `confirmOrCancel` determines that if a `meeting` object receives the `characterize()` message, eventually, and within one day, it should either get the message `confirm()` or the message `cancel()`. Given an instrumented Java™ implementation of these objects and a program in which they exchange messages through method calls, EEAT is able to monitor and assert this invariant at runtime.

Listing 2.1: Example of OCL_{TM} constraint on the Meeting Scheduler.

```
1 context Meeting
2   -- A meeting is either confirmed or canceled within 1 day.
3   def: charact: LTL::OclMessage = receivedMessage('characterize')
4   def: confirm: LTL::OclMessage  = receivedMessage('confirm')
5   def: cancel: LTL::OclMessage  = receivedMessage('cancel')
6   inv confirmOrCancel: after@1d(eventually(charact <> null),
7     eventually((confirm <> null) or (cancel <> null)))
```

As will be shown in later chapters of this thesis, we have chosen OCL_{TM} as the language to specify the monitoring (awareness) requirements for our feedback loops and we have used EEAT to operationalize this monitoring at runtime. In other words, EEAT was used to provide the monitoring of *indicators of requirements convergence* shown in Figure 2.8.

[7] `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.

2.1.7 Qualitative Reasoning

In sub-section 2.1.5, we have represented adaptive systems as feedback control systems, in which requirements are monitored for their satisfaction (as discussed above) and adaptation actions are sent to the target system by some kind of controller or framework. As already mentioned earlier, one of these possible adaptation actions is to change the value of some configuration parameters of the system in order to affect the output in the desired way.

When designing adaptive systems at the requirements level, the system is not yet implemented and its behavior is not completely known. With this incomplete information, we are unable to fully identify how system configuration parameters affect outputs (a fundamental information for the type of adaptation defined as *reconfiguration* in Chapter 1). For this reason, quantitative approaches cannot be applied and, therefore, we base the approach presented in this thesis on ideas from Qualitative Reasoning.[8]

According to Forbus [2004], Qualitative Reasoning is "the area of artificial intelligence (AI) that creates representations for continuous aspects of the world, such as space, time, and quantity, which support reasoning with very little information". In this field, there is a spectrum of choices of qualitative representation languages, each of them providing a different level of *precision* (sometimes referred to as *resolution*). Some examples of qualitative quantity representation languages are [Forbus, 2004]:

- *Status abstraction*: represents a quantity by whether or not it is normal;
- *Sign algebra*: represents parameters according to the sign of their underlying continuous parameter — positive (+), negative (−) or zero (0). It is the weakest form of representation that supports some kind of reasoning;
- *Quantity space*: represents continuous values through sets of ordinal relations, providing variable precision as new points of comparison are added to refine the space;
- *Intervals*: similar to quantity space representation, consists of a variable-precision representation that uses comparison points but also includes more complete information about their ordinal relationship;
- *Order of magnitude*: stratify values according to some notion of scale, such as hyperreal numbers, numerical thresholds or logarithmic scales.

As can be seen in the above examples, the key feature of qualitative reasoning methods is that while frequently there is not enough information to construct quantitative models,

[8] In effect, the use of qualitative information is quite common in Requirements Engineering approaches. An example of such use is the NFR Framework [Mylopoulos et al., 1992].

qualitative models can cope with uncertain and incomplete knowledge about systems and still provide value by allowing some kind of reasoning over the produced models. In other words, they do not require assumptions beyond what is known. Most qualitative reasoning approaches can be seen as having two types of abstraction: domain and functional.

Domain abstraction abstracts the real domain values of variables into a finite number of ordered symbols that describe qualitative values — *landmarks* [Kuipers, 1989] — that are behaviorally significant. Landmarks can be numeric or symbolic and can include the values such as 0 and $\pm\infty$. A qualitative variable value is either a landmark or an interval between adjacent landmarks. The finite, totally ordered set of all the possible qualitative values of a variable composes its *quantity space*.

Qualitative *functional abstraction*, which gives the ability to represent incompletely known functional relationships between quantities, complements domain abstraction in Qualitative Reasoning. For instance, as already mentioned above, signs (+, −, 0) can be used to describe and reason about the direction of change in variables — one can state that there exists some monotonically increasing function relating two quantities, without elaborating further.

Merging qualitative information frequently results in ambiguity, such as when combining positive and negative influences without knowing their magnitudes. The role of ambiguity is important, as it reminds us that further action is necessary, in the form of information gathering and analysis, to increase the level of precision in order to resolve it. In other words, the initial low-level precision of our representations "reveals what the interesting questions are" [Forbus, 2004], which is very useful when eliciting requirements for a system.

Qualitative Reasoning plays a very important role in our approach, as will be shown in the remainder of this thesis.

2.2 Related work

In a recent survey article, Salehie and Tahvildari [2009] propose a taxonomy of adaptation, provide a landscape of the research on adaptive systems, applying the taxonomy to research projects based on their impact in the area and the novelty and significance of their approach. The authors conclude the paper with challenges for this research area based on the comparison between different, existing proposals.

In our research group, Fabiano Dalpiaz [2011] has also published a review of the state-of-the-art in adaptive systems as part of his PhD dissertation, organizing proposals in eight different categories, namely: conceptual models, programming frameworks, software architectures, service-oriented, requirements engineering, algorithms and policies, agent

reasoning and planning and self-organization.

Other survey papers might focus on specific aspects of adaptation. For instance, Di Nitto et al. [2008] reviews the progress if software engineering that led to the concepts in service-oriented computing and Service-Oriented Architectures (SOA), discusses the requirements for self-adaptive systems in the context of service-oriented technologies and points out possible future evolution of this field.

Given the availability of literature reviews for our chosen area of research, our focus in this section will be to present related work that has been analyzed during the time we have worked on the proposals contained in this thesis. In Chapter 8, after we present our approach for the design of adaptive systems, we come back to the research summarized here in order to compare it to our approach.

We start with the research done in the context of Autonomic Computing (Section 2.2.1) and other approaches that focus on architectural issues (Section 2.2.2). Then, we focus on approaches that are more similar to our own, which are the ones that propose the use of requirements models for the design of adaptive systems (Section 2.2.3). Finally, we also touch on the subject of requirements evolution (Section 2.2.4).

2.2.1 Research on Autonomic Computing

Motivated by IBM's autonomic manifesto [Horn, 2001], the past decade has seen a lot of research on Autonomic Computing. As discussed back in Section 1.2.1 (p. 5), some researchers consider *autonomic* and *self-adaptive* as synonyms, whereas others establish differences between their scope and focus. Our research originally started in the context of Autonomic Computing (see, e.g., [Souza and Mylopoulos, 2009]).

An autonomic system, as described by Kephart and Chess [2003], is made of interactive collections of autonomic elements delivering services to users and to other elements according to specified goals and constraints. Each autonomic element consists of one or more managed (software or hardware) resources and an autonomic manager, which controls interactions and the internal state of the element. They must be specified in a standard format and be able to locate and negotiate services they need while provisioning services that are needed from them.

In a 2003 issue of the IBM Systems Journal, Ganek and Corbi [2003] define self-configuring, self-healing, self-optimizing and self-protecting as the cornerstones of autonomic systems self-managing capabilities. Moreover, they proposed a gradual evolution of current systems, starting from a basic level of management and moving towards managed, predictive, adaptive and, finally, an autonomic level. In that same issue [Ritsko, 2003], many other papers provided a glimpse into the research efforts that were being carried out towards autonomic computing.

To provide all of the above *self-** properties, autonomic systems (like adaptive systems in general) base themselves on a feedback loop architecture, called the MAPE-K loop [Kephart and Chess, 2003]. Present in every autonomic manager, this loop performs monitoring, analysis, planning and execution of actions, based on knowledge about the environment, policies, etc., in order to achieve the purpose of self-management of the autonomic element.

More recently, Huebscher and McCann [2008] published a literature review of the major contributors to the aforementioned components of the MAPE-K loop and how this research matches the degrees of autonomicity defined in [Ganek and Corbi, 2003]. The review describes research proposals for the full MAPE-K architecture, monitoring infrastructures, planning models, policies, architectural models and knowledge representation.

2.2.2 Architectural approaches for run-time adaptation

Architecture-based approaches to adaptive systems assume the requirements for the system-to-be — both "vanilla" requirements and those concerned with the adaptation capabilities of the system — are given, and thus concentrate on helping designers build architectures that promote the adaptation features needed by the system. They usually propose the use of an architectural model that shows system components and how they communicate amongst themselves through connectors [Huebscher and McCann, 2008].

The architecture proposed by Autonomic Computing researchers fall into this category, but architectural approaches for adaptive systems have been around even before the publication of the autonomic manifesto. For instance, Oreizy et al. [1999] proposed one such approach, based on an infrastructure that relies on software agents, explicit representation of software components and the environment, plus messaging and event services that coordinate the adaptation. By abstracting the source code of the system into components and their interconnections, adaptation can be performed in a higher level of abstraction (the architecture) and automatically be reflected in the system's implementation.

Another well-known architecture-based approach is that of Kramer and Magee [2007]. They propose a reference architecture for self-adaptive systems based on the three-layer architecture for robotics proposed by Gat [1998]. At the bottom, the *Component Control* layer reports events and status to the upper layer and supports modification of current component configuration. The middle layer, *Change Management*, perform changes in the bottom layer based on situations reported by the latter or new goals introduced by it. It also relies on the top layer in case a situation is reported by the bottom layer and no plans are currently available to tackle it. At the top, the *Goal Management* layer produces change management plans in response to requests from the middle layer or the introduction of new goals.

Subsequent papers built on this architecture proposing the use of modes for service-oriented architectures (SOA) [Foster et al., 2009] and a reactive planning from abstract goals in the top layer together with a plan interpreter and configuration generator in the middle layer [Sykes et al., 2007, 2008]. In the extended approach, the essential characteristics of the environment are represented in a domain model, captured by a finite state machine, which is later transformed to a Labeled Transition System that uses fluents to specify properties of the environment and Linear Temporal Logic to constraint system and environment actions. Based on this model, reactive plans drive adaptation in the change management layer. A case study using this approach is presented in [Heaven et al., 2009] and, more recently, the approach has been further extended to consider non-functional preferences when generating plans for architecture adaptation [Sykes et al., 2010].

The Rainbow framework [Garlan et al., 2004; Cheng et al., 2009c] also uses an architectural model as centerpiece for adaptation. Adaptation rules monitor operational conditions for the system and define actions if the conditions are unfavorable. The key feature of the framework is the use of architectural styles that allows designers to specialize the framework to specific application domains, defining style-specific architectural operators and repair strategies [Garlan et al., 2003]. 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, making the result of this analysis available to consumers who can, for instance, make repair decisions.

Architecture-based approaches usually employ some kind of Architecture Definition Language (ADL) in their models. Kramer and Magee use the Darwin ADL and the Alloy language, specifying components, service ports and interface types [Georgiadis et al., 2002]. The Rainbow framework uses the ACME ADL, which extends the usual component/port representation with the concept of *families*, that allows designers to define architectural styles [Schmerl and Garlan, 2002]. The framework also uses a language called Stitch that aims to "capture routine human adaptation knowledge as explicit adaptation policies", specifying what, when and how to adapt, automating the adaptation process [Cheng, 2008, Chapter 4].

Another architecture-based framework was proposed by Sousa et al. [2009], focusing on allowing users to control Quality of Service (QoS) trade-offs and coordinate the use of resources in a distributed environment composed of several applications. Utility functions for each QoS dimension express user preferences in terms of thresholds for satiation and starvation. Based on the combined utility of each QoS aspect, the Aura Environment Manager [Garlan et al., 2002] was extended in order to compute the optimal resource allocation for each application.

The SASSY framework [Menasce et al., 2011] also focuses on QoS tuning, targeting ser-

vice oriented systems. The approach uses a BPMN[9]-based language called *Service Activity Schema* (SAS) to represent the correct behavior of the system, allowing domain experts to annotate such model with QoS goals. Based on the annotated SAS, the framework generates the system architecture (using xADL — eXtensible Architecture Description Language) selecting the most suitable service provider based on QoS architectural patterns. When QoS violations are detected, the system generates a new architecture and coordinates the process of switching to it at runtime in order to adapt.

As will be seen in more detail in the following chapters, the only contribution towards the system architecture from our approach is the assumption that adaptation will be operationalized by a feedback control loop. We are not alone in this choice. Taylor and Tofts [2004] claim that "self-managed systems are actually closed loop control systems", also recognizing the challenge of directly applying Control Theory methods to complex software systems (cf. Section 1.2.2, p. 7). Their proposal is, thus, to limit the set of measure-response functions in the system so they are known to have the desired properties at all times.

Laddaga and Robertson [2004] also recognize the control paradigm as useful for the design of adaptive systems and propose that systems of this kind should be treated at runtime like a factory, with inputs and outputs, and a control facility that manages it. The functionalities of the controller, such as evaluation, measurement and control, would be developed separately and plugged into the application to manage its reconfiguration. The authors also recognize the planning paradigm as useful for adaptive systems development, but describe it also as a loop including four activities: plan, execute, monitor and revise.

As stated before, the difference between our approach and the ones mentioned in this sub-section is their focus: these approaches propose adaptation at the architectural level, whereas ours is concerned with the requirements for the feedback loop that operationalizes the adaptation. In this way, a system can adapt not only by changing/adjusting components and other architectural elements but actually do it at a higher level of abstraction, adjusting parameters connected to system requirements or changing the system requirements altogether. In this sense, requirements and architecture-based approaches are orthogonal and could be used in combination given the proper transition process (which, unfortunately, is not something that is addressed in this thesis).

2.2.3 Requirements-based approaches for the design of adaptive systems

Like the approach we present in this thesis, some other research proposals on the design of adaptive systems also focus on requirements. The common trait in these proposals is the

[9]Business Process Model and Notation, see `http://www.bpmn.org/`.

consideration of adaptation capabilities of the system during Requirements Engineering, propagating such considerations throughout the software development process.

In what follows, we describe research on adaptive systems that fits the category of requirements-based approaches. As mentioned earlier, a comparison between these approaches and ours is provided in Chapter 8.

RELAX and LoREM

The RELAX language [Whittle et al., 2009, 2010] aims at capturing uncertainty declaratively with modal, temporal, ordinal operators and uncertainty factors provided by the language. RELAX is aimed at capturing uncertainty in the way requirements can be met, mainly due to environmental factors. Unlike goal-oriented approaches, RELAX assumes that structured natural language requirements specifications, containing the SHALL statements that specify what the system ought to do, are available before their conversion to RELAX specifications. The modal operators available, SHALL and MAY...OR, specify, respectively, that requirements must hold or that there exist requirements alternatives (variability).

In RELAX, points of flexibility/uncertainty are specified declaratively, thus allowing designs based on rules, planning, etc. as well as to support unanticipated adaptations. Some requirements are deemed invariant — they need to be satisfied no matter what. Other requirements are made more flexible in order to maintain their satisfaction by using "AS POSSIBLE"-type RELAX operators (e.g., "AS EARLY AS POSSIBLE", "AS CLOSE AS POSSIBLE", etc.). Because of these, RELAX needs a logic with built-in uncertainty to capture its semantics. The authors chose Fuzzy Branching Temporal Logic for this purpose. It is based on the idea of fuzzy sets, which allows gradual membership functions. Temporal operators such as EVENTUALLY and UNTIL allow for temporal component in requirements specifications in RELAX.

In a separate thread of research, Zhang and Cheng [2005, 2006] argue that the semantics for adaptive software should be explicitly captured at the requirements level and, to that purpose, they introduce an extension of Linear Temporal Logic (LTL) called Adapt operator-extended LTL. They propose a 6-step approach for the development of adaptive systems that models global invariants and different domains that the system can operate, then constructs adaptation models from one domain to another. In this approach, adaptive systems are considered to be a collection of steady-state systems and adaptation consists of a dynamic transition from the currently active steady-state system, called *source system*, to another steady-state system, called *target system*.

Here, it is important to note that this definition of *target system* is very different from the one we use throughout this thesis, which is borrowed from Control Theory (as

mentioned back in Section 1.2.2, p. 7). In our approach, *target system* refers to the system that is controlled by the feedback loop, the latter providing adaptation capabilities to the former. Except when referring to approaches based on the work of Zhang and Cheng, in this thesis the term *target system* should be understood as described in this paragraph.

Based on [Zhang and Cheng, 2006], Brown et al. [2006] encapsulate the A-LTL specifications in KAOS models for a more intuitive and graphical representation, allowing the system to switch between operational domains. Later on, Goldsby et al. [2008] proposed the LoREM approach, which defined a systematic processes for performing (Goal-Oriented) Requirements Engineering for adaptive systems. Its name comes from the work of Berry et al. [2005], who defined four *Levels of RE for Modeling* adaptive systems.

In level 1, system developers identify the goals of the system and the steady-state systems that are suitable for the domains that satisfy the goals. Then, in level 2, adaptation scenario developers creates the set of adaptation scenarios, which represent the run-time transitions between source and target systems, including the requirements for monitoring, decision-making and adaptation. Level 3 is concerned with identifying the adaptation infrastructure necessary to support the previously identified scenarios. Finally, level 4 comprises the research done by the community to improve the methods and techniques used in the other levels.

Finally, Cheng et al. [2009a] integrated LoREM and RELAX, adding to the mix an approach to systematically explore the uncertainty form the environment to which the adaptive system will be deployed using threat modeling in KAOS. When a goal threat is identified, there are three possible mitigation strategies that can be applied: (a) add subgoals to handle the condition of the threat; (b) use RELAX to add flexibility to the goal definition; (c) create new high-level goals that capture the objective of correcting the failure. The last strategy works like a feedback loop that adapts the system whenever the goal fails at runtime.

FLAGS

A similar approach to RELAX is FLAGS [Baresi and Pasquale, 2010; Baresi et al., 2010], which proposes crisp (Boolean) goals (specified in LTL, as in KAOS), whose satisfaction can be easily evaluated, and fuzzy goals that are specified using fuzzy constraints. In FLAGS, fuzzy goals are mostly associated with non-functional requirements. The key difference between crisp and fuzzy goals is that the former are firm requirements, while the latter are more flexible.

To provide semantics for fuzzy goals, FLAGS includes fuzzy relational and temporal operators. These allow expressing requirements such as something be almost always less than `X`, equal to `X`, within around `t` instants of time, lasts hopefully `t` instants,

etc. Whenever a fuzzy membership function is introduced in FLAGS, its shape must be defined by considering the preferences of stakeholders. This specifies exactly what values are considered to be "around" the desired value.

Additionally, in FLAGS, adaptive goals define countermeasures to be executed when goals are not attained, using Event-Condition-Action rules. The approach allows for the definition of adaptive goals which, when triggered by a goal not being satisfied, execute a set of adaptation actions that can change the system's goal model in different ways — add/remove/modify goals or agents, relax a goal, etc. — and in different levels — in transient or permanent ways.

At the infrastructure level, Pasquale [2010] proposes an operationalization using a service-oriented architecture. The augmented KAOS models are translated to a *functional model*, composed of variables, activities and messages exchanged by services. Then, a *supervision model* is created, containing directives for monitoring and adaptation of the system, based on the degree of goals' satisfaction. To guarantee safety, adaptations are performed in specific execution points, called *quiescent states* [Zhang and Cheng, 2006].

Approaches based on $i^\star$/Tropos

$i^\star$ and Tropos, introduced earlier in Section 2.1.1, have been extended to represent requirements for system adaptation. Since agents are a very important component in the foundation of these approaches, proposals that extend $i^\star$/Tropos usually keep the focus on the interaction between different agents with one another and with the surrounding environment while pursuing their goals. For an overview of the area, Morandini [2011] provides a review of the state-of-the-art in multi-agent systems in his PhD thesis.

Morandini et al. [2008, 2009] propose extensions to the architectural design phase of the Tropos methodology [Giorgini et al., 2005] to model adaptive systems based on the Belief-Desire-Intention (BDI) model as a reference architecture [**?**]. The approach is called Tropos4AS (Tropos for Adaptive Systems) and introduces new goal types — namely, maintain-goals, achieve-goals and perform-goals — and a new *inhibit* relation between goals that specifies that a goal (the inhibitor) has to be stopped in order for another goal (the inhibited) to be achieved/maintained.

Tropos4AS also extends Tropos in order to allow designers to model non-intentional elements using UML[10] class diagrams, specifying resources that belong to an agent and the ones that belong to the environment. The approach also allows for the modeling of undesirable (faulty) states, which are known to be possible at runtime and should trigger system adaptation. Finally, Morandini [2011] maps the goal models to the Jadex[11]

[10] The Unified Modeling Language, see `http://www.uml.org/`.

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

platform for run-time implementation.

Ma et al. [2009] use $i^\star$ and the NFR framework to represent preferences, which ultimately drive service selection in service-oriented applications. NFR constructs are used to model the interrelation among different criteria for service selection according to domain experts, but the authors use quantitative values instead of the usual, qualitative labels of NFR. $i^\star$ actor dependencies are also used to represent alternative services networking decisions. The authors provide algorithms that reason over these two kinds of models to identify optimal solutions.

Another approach is the one from Dalpiaz et al. [2009, 2010, 2012], which proposes an architecture that, based on requirements models, adds self-reconfiguring capabilities to a system using a monitor-diagnose-compensate (MDC) loop. A *monitor* component collects, filters and normalizes events/logs from the system, which serve as input to the *diagnose* component, responsible for identifying failures and discovering their root causes. Finally, the *reconfigurator* component selects, plans and deploys compensation actions in response to failures.

The authors propose different algorithms for system reconfiguration at runtime. One such algorithm finds all valid variants to satisfy a goal and compares them based on their cost (to compensate tasks that failed or the ones that already started and will be canceled) and benefit (e.g., contribution to softgoals) [Dalpiaz et al., 2012]. Another algorithm reconfigures the system in terms of interaction among autonomous, heterogeneous agents based on commitments, proposing different adaptation tactics, such as exploiting variability, goal/commitment redundancy, switching debtors, division of labor, etc. [Dalpiaz et al., 2010].

The Continuous Adaptive Requirements Engineering (CARE) method [Qureshi and Perini, 2009, 2010; Qureshi et al., 2011b] is also based on Tropos, focusing on service-based applications. At design time, developers specify *adaptive requirements* along with "vanilla" requirements using goal models. Adaptive requirements take into account not only functional and non-functional concerns but also monitoring and variability. Domain ontologies represent the knowledge about the domain and are linked to the goal model to help analysts detail the expected behavior of the system.

Based on these models, the system monitors for environmental changes (that violate goals) or user requests (queries), representing them as *Run-time Requirement Artifacts* (RRAs), which consist of service requests. Whenever an RRA is acquired, lookup is performed in order to find a service that satisfies the request. Service selection can be done automatically based on user preferences or manually by the user herself. Finally, the system's specification is modified, adding the selected service and possibly removing others.

In [Qureshi et al., 2011a], the authors formulate a runtime requirements adaptation problem for self-adaptive system and extend the Core ontology for requirements [Jureta et al., 2008] with the concepts of *Context* and *Resource* and new relations *Relegation* and *Influence* between requirements. The former establishes that if a requirement cannot be satisfied in one way, other less-preferred ways can be tried, whereas the latter indicates how the satisfaction of one requirement influences in the satisfaction of another.

Reconfiguration approaches

In the context of GORE, Wang and Mylopoulos [2009] define a system *configuration* as "a set of tasks from a goal model which, when executed successfully in some order, lead to the satisfaction of the root goal." In Chapter 1, we have called *reconfiguration* the act of searching the solution space for parameters (e.g., the choice of the path to take in OR-refinements, which determines the set of tasks to be executed) that can be changed in order to improve the system's outcome. Below, we present some approaches that propose adaptation through reconfiguration.

Hawthorne and Perry [2004] propose a prescriptive architecture for self-adaptive systems called Distribution Configuration Routing (DCR). This KAOS-based approach starts with goal-oriented requirements engineering, specifying object *roles*, whose behavior is specified by *intents*. DCR is then able to compose system configurations that are conformant with requirements by analyzing role and intent models.

Brake et al. [2008] automate the discovery of software tuning parameters at the code level using reverse engineering techniques. A taxonomy of parameters and patterns to aid in their automatic identification provides some sort of qualitative relation among parameters, which may be "tunable" or just observed. The approach targets existing and legacy software, compiling an initial catalog of parameters by analyzing the system's documentation, then executing a syntactical search of the source code to find fields that match the identified parameters.

Khan et al. [2008] apply Case-Based Reasoning to the problem of determining the best system configuration. System configurations are kept in the case-base as solutions and associated to problems cataloged from past experience. At runtime, when problems are detected, an algorithm searches for a solution in the case-base using a similarity measure. Another algorithm evaluates if the problem is new (and should be cataloged) or if it can be associated with a template in order to restrict the fast growth of the case-base, which would make run-time adaptation more difficult.

Wang and Mylopoulos [2009] propose algorithms that suggest a new configuration without the component that has been diagnosed as responsible for the failure. Their framework receives as input a goal model, in which each goal/task is given a precondition,

an effect and a monitor status. Preconditions and effects are propositional formulas representing conditions that must be true before and after, respectively, a goal is satisfied or a task is executed, whereas the monitor status indicates if a task or goal should be monitored or not, making it possible to control the desired granularity level of diagnostics.

The monitoring layer instruments the source code of the program in order to provide the diagnostic layer a log (truth values for observed literals or the occurrence of a task at a specific time-step). The diagnostic layer can then produce axioms for *deniability* (a task or goal occurred but either its precondition or its effect did not), *label propagation* (propagate satisfiability and deniability between tasks and subgoals towards their parent goals) and *contribution* (calculate the effect that contribution links have on their targets based on the satisfiability or deniability of the source goal/task). Axioms and log entries are encoded and passed to a SAT solver, which translates them into diagnoses.

Fu et al. [2010] represent the life-cycle of instances of goals at runtime using a state-machine diagram and, based on it, an algorithm can prevent possible failures or repair the system in case of requirements deviation. Coupled with event mapping rules in first-order logic, the state-machine diagram specifies in detail the traceability between runtime and requirements, allowing the system to reconfigure based on a standard set of activities (e.g. retry, propagate, try a different path in an OR-refinement, etc.). The reconfiguration policies can be associated with *use limits* (specifying the upper bound for the execution of a given policy) and *avoidance goal state patterns* (regular expression-like pattern that rules out the policy if the monitored goal's history matches it).

Like our own work, Peng et al. [2010] propose an adaptation approach founded on goal reasoning and feedback control theory (using the PID controller). A proposed procedure receives as input a goal model with softgoals ranked by preference and finds a configuration of the system (i.e., a set of leaf-level tasks, in the spirit of the NFR Framework) that optimizes the achievement of of high-ranked softgoals. In practice, a SAT solver is used to try and find a configuration that accommodates all soft-goals. If that can't be done, drop the lowest-ranked softgoal and try again, proceeding iteratively this way until a configuration can be found (or all softgoals have been dropped). Architectural reconfigurations are supported by a SOA and a reflective component model.

Moreover, modification of softgoal preference ranks are allowed at runtime. The control input for the feedback loop is some business value that has to be reached. At runtime, if a business value associated with a specific softgoal (e.g., value "response time" and softgoal "minimal response time") is below some threshold, the rank of the associated softgoal is increased and the reconfiguration procedure is executed. The authors adapt existing qualitative goal reasoning frameworks (e.g., [Giorgini et al., 2003]) to business value propositions that are quantitative.

Nakagawa et al. [2011] developed a compiler that generates architectural configurations by performing conflict analysis on KAOS goal models. The compiler reads the models and generates architectural configurations for self-adaptive systems. Such configurations use multiple control loops based on an extension introduced in KAOS. Goal elements are divided in 3 categories, one for each part of the control loop: *monitor*, *analyze & decide*, and *act*.

Salehie and Tahvildari [2012] propose GAAM, the Goal-Action-Attribute Model. In this approach, measurable/quantifiable properties of the system are modeled as *attributes*, whereas goals are represented in their usual, hierarchical way. Goals are assigned *weights/priorities* and the model also keeps track of each goal's *activation level*. Moreover, changes that are applicable to adaptable software entities are modeled as *adaptation actions* and a *preference matrix* specifies their order of preference toward goal satisfaction. Finally, an *aspiration level matrix* determines the desired levels of attributes of each goal. At runtime, polling monitors attributes and goals, comparing them to the aspiration levels, and an action selection mechanism based on goal weights reconfigures the system in case an attribute does not reach its aspired level.

Previously described approaches, such as the ones from Morandini et al. [2009] and Dalpiaz et al. [2012], can also be considered reconfiguration approaches, in their case focusing on agents, their goals and social relations. Recently, Ali et al. [2011b] also explored the role of social relationships among system users when deciding how to adapt a system.

Design-time trade-off analysis and risk management

Although not explicitly designed for run-time system adaptation, approaches that propose design-time trade-off analysis or risk management could be adapted to be used at runtime in order to decide the best system configuration. The former analyzes alternatives to choose the best one for a given problem, whereas the latter is concerned with modeling things that can go wrong with a software system, both of which are activities that adaptive systems have to perform at runtime.

Letier and van Lamsweerde [2004] present an approach that allows for specifying partial degrees of goal satisfaction for quantifying the impact of alternative designs on high-level system goals. Their partial degree of satisfaction can be the result of, e.g., failures, limited resources, etc. and is measured in terms of the probability that the goal is satisfied. Thus, the approach augments KAOS with a probabilistic layer.

Here, goal behavior specification (in the usual KAOS temporal logic way) is separate from the quantitative aspects of goal satisfaction: domain-specific *quality variables* associated with goals are modeled and *objective functions* define goal-related quantities to

be maximized or minimized. An approach for propagating partial degrees of satisfaction through the model is also part of the method, allowing one to determine the degree of satisfaction of a goal from the degrees of satisfaction of its subgoals. Finally, alternative designs can be evaluated and compared by computing the objective functions.

More recently, Heaven and Letier [2011] propose the use of stochastic simulations to generate sample values for each leaf-level quality variable according to its probability distribution in order to compute the objective functions obtained in the simulation. With this simulation, manual comparison between distinct design choices can be performed, but this is not easy and alternatives can grow exponentially with the increase of variability points. Therefore, the authors propose to solve this multi-objective optimization problem by finding the set of Pareto-optimal solutions (i.e., the ones that are not dominated by other solutions and thus can be compared in a trade-off analysis) and using meta-heuristic search algorithms (e.g., exhaustive search for small problems, genetic algorithms for large ones, etc.).

The Defect Detection and Prevention (DDP) process [Cornford et al., 2006] was proposed to achieve life-cycle risk management in software projects. The process starts by capturing the system requirements using a tree structure. Then, possible situations in which the requirements are not achieved are analyzed and represented in a tree of potential failure modes, which is prioritized based on the impact they have on requirements. Finally, the developers devise *Preventative measures, Analyses, process Controls and Tests* (PACTs) to mitigate the identified failures, aiming at minimizing the overall risk to the project.

Menzies and Richardson [2006] propose to simulate (execute) qualitative models of requirements to explore scenarios and learn from the models before spending resources detailing them and developing the system. Since qualitative models generate scenarios exponentially to the number of variables, the authors propose the identification of *master variables*, i.e., key parameters that set the value of remaining slave variables. Then a tool called TAR3 performs stochastic forward select (stochastic simulation/sampling) to search for treatments (treatment learning, a treatment being a setting to the master variables that improves the performance of the qualitative models).

According to the approach, after reading the early requirements, software process options are listed as Boolean parameters. Their effect on interesting indicators are specified qualitatively (+, −, 0, ?) by experts. Stakeholders also assign utility values to each indicator and a formula calculates the overall utility. Key parameters are identified and simulation finds what is the value they should have in order to improve the indicators as much as possible.

The proposals of Elahi and Yu [2011] on design-time trade-off analysis — pair-wise

comparison of alternatives with respect to goals that were selected as indicators — could be adapted to provide information for run-time adaptation if we could somehow remove the need for stakeholder intervention in the analysis. For instance, contribution links in $i^\star$ can provide qualitative relations between alternatives and monitored indicators, although they lack the means of differentiating between links with the same label (GRL provides numeric contribution values and, thus, could be used here).

Control-theoretic approaches

In Chapter 1, we have cited a few road-map papers that highlight the need to apply concepts from Control Theory, such as feedback loops, in the design and development of adaptive systems [Brun et al., 2009; Andersson et al., 2009; Cheng et al., 2009b]. This thesis is an effort in this direction, but we are certainly not the only ones.

Schmitz et al. [2008] (see also [Schmitz et al., 2009]) use $i^\star$ goals to model the requirements of control systems. The *target/controlled system*, the *controller* and their combination are considered $i^\star$ actors, then $i^\star$ dependencies capture their relationships. For instance, *resource dependencies* capture sensors and actuators. The approach focuses on reuse of software artifacts when designing control systems, providing an automatic identification of potentially reusable components. Moreover, the authors propose a semi-automatic process to derive mathematical models commonly used in control systems development from the requirements.

Hebig et al. [2010] present a UML profile for the creation of architectural models that represent control loops as first-class citizens. The profile defines roles for *process component*, *controller*, *sensor* and *actuator* components, the latter three also featuring in interface stereotypes that establish relationships among the different components. Moreover, *strands* define when control interfaces in the system are intended to be influenced by an actuator, whereas *effect propagation* indicates that a change in the input of a component leads to a change in the output of the component. *Sensor interfaces* can also define the scope of the control loop by marking it as *controllable* or *environmental*. Multiple loops are also supported.

Filieri et al. [2011] applied control theory to the problem of designing adaptive systems with a requirements perspective, focusing on adapting to failures in reliability and modeling requirements using Discrete Time Markov Chains (DTMCs). There, transitions are labeled with control variables, whose values can be set by a controller that decides the system's settings in order to keep satisfying the requirements. Well established control theoretic tools are used to design such controller and the authors claim the approach can be extended to deal with failures of different nature. This approach is extended in [Filieri et al., 2012] with a more efficient solution for dynamic binding of components and an

auto-tuning procedure.

2.2.4 Requirements evolution

The problem of requirements evolution was initially addressed in the context of software maintenance, focusing on maintaining the requirements models synchronized with their implementation once the system goes into maintenance. To cite a few examples, in [Wenjie and Shi, 2009] a method based on Π-Calculus and OWL-S[12] is proposed for efficient and controllable software evolution. Revolution2 [Duan, 2009] uses refined use cases and refactoring techniques to propagate changes in use cases to subsequent models.

The approach proposed by Ben Charrada and Glinz [2010] analyses changes in test suites and provides hints for updating the requirements specifications. Villela et al. [2008] propose a method for identification of unstable features and anticipation of potential adaptation needs for embedded systems, which is, according to the authors, easily generalizable for software systems. Their method provides only designer-supported evolution, which is based on analysis provided by domain and market experts in order to anticipate the adaptation needs.

The topic has also been recently gaining attention from the Requirements Engineering research community. In a chapter of a recent book entitled "Design Requirements Engineering: A Ten-Year Perspective", Ernst et al. [2009] discuss the state-of-the-art for research on the topic, and predict some of the research problems for the next 10 years. The authors also provide a concrete proposal for a run-time monitoring framework based on requirements. This is later extended to automate reconciliation in high-variability systems by reconfiguration — [Wang and Mylopoulos, 2009], cited earlier. Another approach, by Nissen et al. [2009], investigates the consequences for the evolution of requirements for control systems and proposes countermeasures to problems caused by this evolution. The approach builds on earlier work by Schmitz et al. [2008], also cited above.

Based on the redefinition of the *requirements problem* by Jureta et al. [2008], Ernst et al. [2011] developed a *requirements engineering knowledge base* that stores the information acquired during requirements elicitation and provides tools for answering various queries, such as, for instance, comparing alternative solutions. Then, the authors define the *requirements evolution problem* as the one of finding a new solution for a requirements problem that has been modified, focusing on reusing as much as possible the existing solution.

Requirements evolution research has also focused on modeling requirements change and its impact on the system. For instance, in Lam and Loomes [1998], environment

[12] See `http://www.w3.org/Submission/OWL-S/`.

changes are propagated through requirements changes and down to design. Each triggered requirements change is analyzed in terms of its risks and the impact it has on the users' needs. Another important aspect of requirement evolution is the completeness and consistency of requirements models. For instance, to address this, Zowghi and Offen [1997] propose a formal approach based to requirements evolution utilizing non-monotonic default logics with belief revision.

2.3 Chapter summary

In this chapter, we have summarized the research that was used as baseline for the proposals in this thesis (§ 2.1). First, our proposal is to design adaptive systems with a Requirements Engineering (RE) perspective and, therefore, we start from the state-of-the-art in RE, namely Goal-Oriented Requirements Engineering (GORE), more specifically the core ontology for RE proposed by Jureta et al. [2008] (§ 2.1.1). By applying existing methods and tools for goal-oriented requirements elicitation and modeling, eventually one can produce a GORE-based requirements specification in the form of a goal model (§ 2.1.3).

Given one such model, other fields of study provide us with tools that are very useful in the process of engineering requirements for adaptive systems. First, variability in requirements, in particular goal models (§ 2.1.4), allows for the representation of different ways of satisfying the system goals at runtime, which is the basis for adaptation through reconfiguration. Second, as stated in Section 1.2.2 (p. 7), adaptive systems contain some kind of feedback loop and, thus, we adopt ideas from the field of Feedback Control Theory (§ 2.1.5). Third, concerning the first step of the feedback loop, existing approaches of requirements monitoring should be harnessed (§ 2.1.6). Last, but not least, given the difficulty in providing precise information about the behavior of a system-to-be during the elicitation of its requirements, approaches from the area of Qualitative Reasoning become very useful and should be considered (§ 2.1.7).

While presenting the foundation for our work, we have also introduced the running example of this thesis, namely the Meeting Scheduler (§ 2.1.2). Such example will be used throughout the following chapters in order to illustrate our proposals.

Finally, this chapter also summarizes different work that is related to the proposals of this thesis (§ 2.2), in particular the proposals of Autonomic Computing (§ 2.2.1), architecture-based approaches (§ 2.2.2), requirements-based approaches (§ 2.2.3) and work on requirements evolution (§ 2.2.4).

Chapter 3

Modeling adaptation requirements

Science is what we understand well enough to explain
to a computer. Art is everything else we do.

Donald Knuth

This chapter details the first contribution of our approach to the design of adaptive systems: given a GORE-based requirements specification (cf. Section 2.1.3, p. 32), how can we model the adaptation requirements for the system, harnessing the abstractions provided by the feedback loop architecture which actually implements the adaptation? In other words, in this chapter we attempt to answer research questions **RQ1** and **RQ2**, stated in Chapter 1: *What are the requirements that lead to the adaptation capabilities of a software system's feedback loop?* and *How can we represent such requirements along with the system's "vanilla" requirements?*[1]

The answer to these questions are presented in the following sections. Section 3.1 proposes *Awareness Requirements* as indicators of what the feedback loop must monitor, defining the criteria for what constitutes a requirements divergence (i.e., a control error). Then, Section 3.2 presents *Evolution Requirements* as a way of representing how the requirements model itself could be changed in order for the system to adapt.

Berry et al. [2005] defined the *envelope of adaptability* as the limit to which a system can adapt itself: "since for the foreseeable future, software is not able to think and be truly intelligent and creative, the extent to which a [system] can adapt is limited by the extent to which the adaptation analyst can anticipate the domain changes to be detected and the adaptations to be performed." In this context, to completely specify a system with adaptive characteristics, requirements for adaptation have to be included in the

[1] Part of these questions will also be answered in Chapter 4, where we present new model elements that promote qualitative adaptation through reconfiguration, and Chapter 6, when we show how to represent the requirements in a machine-readable format for their use in the run-time framework.

specifications.

We propose the aforementioned new classes of requirements to fill this need, promoting feedback loops for adaptive systems to first-class citizens in Requirements Engineering. Considering the feedback loop, *Awareness Requirements* constitutes the requirements for the monitoring component, whereas *Evolution Requirements* represents the requirements for the adaptation component.

This chapter focuses on the modeling elements that represent the requirements for system adaptation, whereas the process through which these requirements are elicited and these models are built is covered in Chapter 5. We expect that the abstractions provided by these new elements will help developers model and communicate adaptation requirements. Furthermore, in Chapter 6 we discuss how these models can be used at runtime by an adaptation framework.

3.1 Awareness Requirements[2]

As previously mentioned, our research started by applying a Requirements Engineering perspective to the feedback loop architecture, studying the requirements that lead to the functionality provided by feedback loops to adaptive systems. In other words, if feedback loops constitute an (architectural) solution, what is the requirements problem this solution is intended to solve?

The nucleus of an answer to this question can be gleamed from any description of feedback loops: "... the objective ... is to make some output, say y, behave in a desired way by manipulating some input, say u ..." [Doyle et al., 1992]. Suppose then that we have a requirement R = "produce meeting schedules upon request" and let S be a system operationalizing R. The "desired way" of the above quote for S is that it *always* fulfills R, i.e., every time there is a request for a meeting the system successfully produces a schedule. Note that, here, the notion of "success" depends on the type of system: for software systems, it means completing the transaction without errors or exceptions, whereas for socio-technical systems "success" could involve the participation of human actors, e.g., the secretary notifies all participants.

In any case, this means that the system somehow manages to deliver its functionality under (almost) all circumstances (e.g., even when not enough participants have responded about their timetables). Such a requirement can be expressed, roughly, as R' = "Every instance of requirement R succeeds". And, of course, an obvious way to operationalize R' is to add to the architecture of S a feedback loop that monitors if system responses to

[2] **Acknowledgment**: an early version of the results presented in this section was included in the PhD thesis of Alexei Lapouchnian [2010], who collaborated also in other parts of the research portrayed in this thesis.

requests are being met, and takes corrective action if they are not.

We can generalize on this: we could require that S succeeds more than 95% of the time over any one-month period, or that the average time it takes to schedule a meeting over any one week period is no more than 1 day. The common thread in all these examples is that they define requirements about the run-time success/failure/quality-of-service of other requirements. We call these *self-awareness requirements*.

A related class of requirements is concerned with the truth/falsity of domain assumptions. For our example, we may have designed our Meeting Scheduler system on the domain assumption D = "there is always at least one room available". Accordingly, if room availability is an issue for our system, we may want to add yet another requirement R'' = "D will not fail more than 2% of the time during any 1-month period". We call these *contextual awareness requirement*, as they are concerned with the truth/falsity of domain assumptions.

To generalize the types of requirements, illustrated by R' and R'', we call them *Awareness Requirements* (hereafter referred to as *AwReqs*). We characterize them syntactically as requirements that refer to other requirements or domain assumptions and their success or failure at runtime. *AwReqs* are represented in an existing language in the system requirements specification and can be directly monitored by a requirements monitoring framework at runtime (the latter is further discussed in Chapter 6).

The above definitions are in line with our view of adaptive systems as feedback control systems, presented in Section 2.1.5 (p. 39). In Control Systems terms, the reference input in this case is the system fulfilling its mandate, i.e., its requirements. Measuring the actual output and comparing it to the reference input is the first step performed by a feedback loop and, in the case of adaptive systems, this amounts to verifying if requirements are being satisfied or not.

Awareness is a topic of great importance within both Computer and Cognitive Sciences. In Philosophy, awareness plays an important role in several theories of consciousness. In fact, the distinction between self-awareness and contextual awareness seems to correspond to the distinction some theorists draw between higher-order awareness (the awareness we have of our own mental states) and first-order awareness (the awareness we have of the environment) [Rosenthal, 2005]. In Psychology, consciousness has been studied as "self-referential behavior". Closer to home, awareness is a major design issue in Human-Computer Interaction (HCI) and Computer-Supported Cooperative Work (CSCW) [Schmidt, 2002]. The concept in various forms is also of interest in the design of software systems (security / process / context / location / ... awareness).

In the following sub-sections, we characterize *AwReqs* in more detail, discuss how to specify them in a language with a higher degree of formality than natural language,

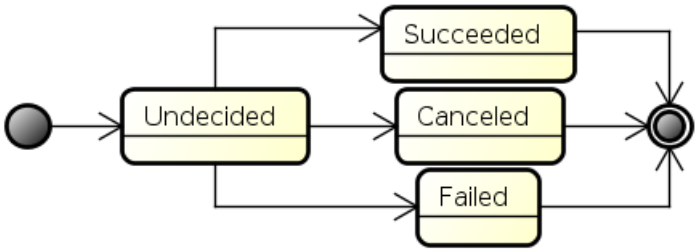


Figure 3.1: States assumed by GORE elements at runtime.

propose the means to adding them to the system's requirements specification through the use of patterns and, if desired, a graphical representation. All of these aspects are illustrated with examples from the Meeting Scheduler, which has been introduced back in sections 2.1.2–2.1.4 (p. 27).

3.1.1 Characterizing *AwReqs*

AwReqs are requirements that talk about the run-time status of other requirements (here, domain assumptions can be considered "requirements on the environment", similar to what is proposed by van Lamsweerde and Willemet [1998], to simplify the general characterization of *AwReqs*). More precisely, *AwReqs* talk about the states requirements can assume during their execution at runtime. We use the expression "requirement execution" to denote the situation in which an actor is pursuing the satisfaction of a requirement through the system. Figure 3.1 shows these states which, in the context of our modeling framework, can be assumed by goals, tasks, domain assumptions, quality constraints (cf. Section 2.1.3 in p. 32, also [Jureta et al., 2008]) and *AwReqs* themselves.

When an actor starts to pursue a requirement, its result is yet `Undecided`. Eventually, the requirement will either have `Succeeded`, or `Failed`. For goals and tasks, which are "long-running, performative requirements", there is also a `Canceled` state. Our approach currently considers only these four states, but could very easily be extended to consider other, new states, increasing the expressiveness of *AwReqs* (however, this would also increase the responsibilities of the target system in terms of indicating changes of states in requirements, as will be explained in Chapter 6).

Table 3.1 shows some of the *AwReqs* that were elicited during our analysis of the Meeting Scheduler. These examples are presented to illustrate the different types of *AwReqs*, which are discussed in the following paragraphs. In other words, we do not claim that this set of *AwReqs* is either necessary or sufficient for satisfying the adaptation requirements of stakeholders of the average Meeting Scheduler.

The examples illustrate a number of types of *AwReq*. *AR1*, *AR4*, *AR5* and *AR7*

Table 3.1: Examples of *AwReqs*, elicited in the context of the Meeting Scheduler.

Id	**Description**	**Type**
AR1	Task *Characterize meeting* should never fail.	–
AR2	Quality constraint *Meetings cost less than €100* should be satisfied 75% of the time.	Aggregate
AR3	The success rate of goal *Collect timetables* should not decrease two weeks in a row.	Trend
AR4	Goal *Find a suitable room* should never fail.	–
AR5	Goal *Choose schedule* should never fail.	–
AR6	Quality constraint *At least 90% of participants attend* should have a 75% success rate per month.	Aggregate
AR7	Domain assumption *Participants use the system calendar* should always be true.	–
AR8	Domain assumption *Local rooms available* should be false no more than once a week.	Aggregate
AR9	Task *Let system schedule* should successfully execute at least ten times as much as task *Schedule manually*.	Aggregate
AR10	Quality constraint *Schedules produced in less than a day* should have 90% success rate over the past ten days, checking daily.	Aggregate
AR11	Goal *Manage meeting* should be satisfied within one hour of the time set by the meeting's schedule.	Delta
AR12	Task *Confirm occurrence* should be decided within five minutes.	Delta
AR13	*AwReq AR7* should succeed 80% of the times.	Aggregate (Meta)
AR14	The monthly success rate of *AwReq AR6* should not decrease twice in a row.	Trend (Meta)

show the simplest form of *AwReq*: the requirement to which they refer should never fail. Considering a control system, the reference input is to fulfill the requirement. If the actual output is telling us the requirement has failed, the control system must act (adapt) in order to bring the system back to an acceptable state.

AwReqs like these consider every instance of the referred requirement. An instance of a task is created every time it is executed and the "never fail" constraint is to be checked for every such instance. Similarly, instances of a goal exist whenever the goal needs to be fulfilled, while domain assumptions and quality constraint instances are created whenever their truth/falsity needs to be checked in the context of a goal fulfillment. Satisfaction of the elements in a GORE-based specification were briefly discussed back in Section 2.1.3 (p. 32).

Inspired by the three modes of control of the proportional-integral-differential (PID) controller (cf. Section 2.1.5, p. 39), we propose three types of *AwReqs*, briefly described below and further illustrated in the following paragraphs:

- *Aggregate AwReqs* act like the integral component, which considers not only the current difference between the output and the reference input (the control error), but aggregates the errors of past measurements;
- *Delta AwReqs* were inspired by how proportional control sets its output proportional to the control error;
- *Trend AwReqs* follow the idea of the derivative control, which sets its output according to the rate of change of the control error.

An *aggregate AwReq* refers to the instances of another requirement and imposes constraints on their success/failure rate. For example, *AR2* is the simplest aggregate *AwReq*: it demands that the referred quality constraint be satisfied 75% of the time the goal *Schedule meeting* is attempted.

Aggregate *AwReqs* can also specify the period of time to consider when aggregating requirement instances, e.g., *AR6* indicates a month as this period. The frequency with which the requirement is to be verified is an optional parameter for *AwReqs*. If it is omitted, then the designer is to select the frequency (if the period of time to consider has been specified, it can be used as default value for the verification frequency). *AR10* is an example of an *AwReq* with period of time (past ten days) and verification interval (every 24 hours) specified.

Another pattern for aggregate *AwReq* specifies the min/max success/failure a requirement is allowed to have. For instance, *AR8* indicates that a specific domain assumption should be false at most once a week. *AwReqs* can combine different requirements, like

AR9, which compares the success counts of two tasks, specifying that one should succeed at least ten times more than the other. This captures a desired property of the alternative selection procedure when deciding at runtime how to fulfill a goal.

AR3 is an example of a *trend AwReq* that compares success rates over a number of periods. Trend *AwReqs* can be used to spot problems in how success/failure rates evolve over time and could be used, for instance, to predict an upcoming undesirable situation, given a negative trend on the success rate. In the example, *AR3* specifies that the success rate of a goal should not decrease twice in a row, considering week periods.

Delta AwReqs, on the other hand, can be used to specify acceptable thresholds for the fulfillment of requirements, such as achievement time. *AR11* specifies that goal *Manage meeting* should be satisfied within one hour of the start of the meeting. Note how, in this case, the *AwReq* refers to a property of an entity of the problem domain (a meeting).[3]

Another delta *AwReq*, *AR12*, shows how we can talk not only about success and failure of requirements, but about changes of states, following the state machine diagram of Figure 3.1. In effect, when we say a requirement "should [not] succeed (fail)" we mean that it "should [not] transition from `Undecided` to `Succeeded` (`Failed`)". *AR12* illustrates yet another case: the task *Confirm occurrence* should be decided — i.e., should leave the `Undecided` state — within five minutes. In other words, regardless if they succeeded or fail, secretaries should not spend more than five minutes confirming if a meeting has occurred or not.

Finally, *AR13* and *AR14* are the examples of *meta-AwReqs*: *AwReqs* that talk about other *AwReqs*. As we have previously discussed (cf. Section 1.2.2, p. 7), *AwReqs* are based on the premise that even though we elicited, designed and implemented a system planning for all requirements to be satisfied, at runtime things might go wrong and requirements could fail, so *AwReqs* are added to trigger system adaptation in these cases. Using the same rationale, given that *AwReqs* themselves are also requirements, it follows that they are also bound to fail at runtime. Thus, meta-*AwReqs* can provide further layers of adaptation in some cases if needed be.

Meta-*AwReqs* also belong to one of the previous categories of *AwReqs*. For instance, *AR13* is an aggregate meta-*AwReq* that specifies that *AR7* should fail no more than 20% of the time. In its turn, *AR14* is a trend meta-*AwReq*, constraining the success rate of *AR6* to not decrease two months in a row.

With enough justification to do so, one could model an *AwReq* that refers to a meta-*AwReq*, which we would call a meta-meta-*AwReq* — or third-level *AwReq*. There is no

[3] Although one can represent such an *AwReq*, it will be seen in Chapter 6 that a limitation of our approach is that it does not currently integrate with domain models and, thus, cannot automatically operationalize *AwReqs* like this. It can, however, operationalize *AwReqs* that compare properties of the requirement objects, e.g., "Requirement R_1 should be satisfied within one hour of the satisfaction of Requirement R_2."

limit on how many levels can be created, as long as meta-*AwReqs* from a given level refer strictly to *AwReqs* from lower levels, in order to avoid circular references. It is important to note that the name meta-*AwReq* is due only to the fact that it consists of an *AwReq* over another *AwReq*. This does not mean, however, that multiple levels of adaptation loops are required to monitor them. As will be presented in Chapter 6, monitoring is operationalized by matching method calls to changes of states of requirements instances, regardless of the class of the object that is receiving the message (goal, task, *AwReq*, meta-*AwReq*, etc.).

3.1.2 *AwReqs* specification

We have just introduced *AwReqs* as requirements that refer to the success or failure of other requirements. This means that the language for expressing *AwReqs* has to treat requirements as first class citizens that can be referred to. Moreover, the language has to be able to talk about the status of particular requirements instances at different time points.

As mentioned in Section 2.1.6 (p. 43), we have chosen to use an existing language, OCL_{TM}, over creating a new one, therefore inheriting its syntax and semantics. The subset of OCL_{TM} features available to requirements engineers when specifying *AwReqs* is the subset supported by the monitoring framework, EEAT, also previously introduced. A formal definition of the syntax and the semantics of *AwReqs* is out of the scope of this thesis.

Our general approach to using OCL_{TM} is as follows: (i) design-time requirements, such as the goal model for the Meeting Scheduler shown in Figure 2.6 (p. 37), but also the *AwReqs* of Table 3.1, are represented as UML classes; (ii) run-time instances of requirements, such as various meeting scheduling requests, are represented as instances of these classes.

Representing system requirements (previously modeled in a goal model) in a UML class diagram is a necessary step for the specification of *AwReqs* in any OCL-based language, as OCL constraints refer to classes and their instances, attributes and methods. Even though other UML diagrams (such as the sequence diagram or the activity diagram) might seem like a better choice for the representation of requirements and *AwReqs*, having instances of classes that represent requirements at runtime is mandatory for the OCL-based infrastructure that we have chosen.

Hence, we present in Figure 3.2 a model that represents classes that should be extended to specify requirements. This model is the result of an analysis of the core ontology for requirements engineering proposed by Jureta et al. [2008] (cf. Section 2.1.3, p. 32), reported in more detail in [Souza, 2010].

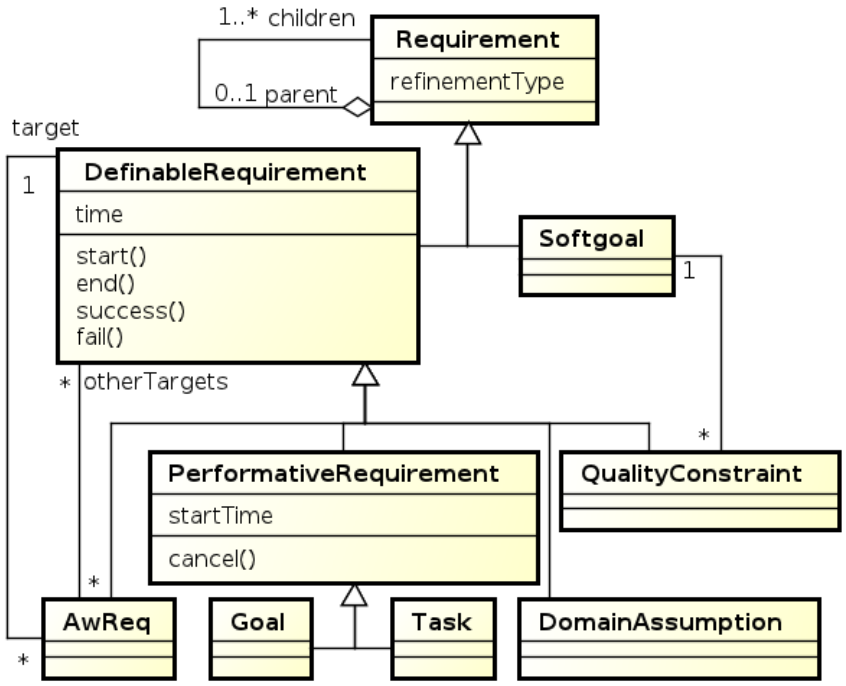


Figure 3.2: Class model for requirements in GORE-based specifications.

Each requirement of the adaptive system should be represented by a UML class, extending the appropriate class from this diagram. Table 3.2 lists the requirements for the Meeting Scheduler (depicted in Figure 2.6, p. 37) and their respective UML class name and super-class from Figure 3.2. To make their identification easier, we use mnemonics for the name of the classes, prepending them with the initial of the extended super-class. Although not shown in the table, the *AwReqs* of Table 3.1 are also represented in UML, using their IDs as class name and extending the `AwReq` class.

Note that the diagram of Figure 3.2 does not represent a *meta-model* for requirements due to the fact that the classes that represent the system requirements are subclasses of the classes in this diagram, not instances of them as it is the case with meta-models. This inheritance is necessary in order for *AwReq* specifications to be able to refer to the methods defined in these classes, which are inherited by the requirement classes.

Another important observation is that these classes are only an abstract representation of the elements of the goal model, being part of the architecture of the monitoring framework (that will be presented in Chapter 6) and not of the target system's implementation (i.e., the Meeting Scheduler itself). In other words, the actual requirements of the system are not implemented by means of these classes.

Listing 3.1 shows the specification of the *AwReqs* of Table 3.1 using OCL_{TM}. For example, consider *AR1*, which refers to a task requirement. In the listing, *AR1* is specified

Table 3.2: Meeting scheduler requirements and their UML representations.

Requirement	**UML Class Name**	**Super-class**
Schedule meeting	G_SchedMeet	Goal
Characterize meeting	T_CharactMeet	Task
Collect timetables	G_CollectTime	Goal
Call participants	T_CallPartic	Task
Email participants	T_EmailPartic	Task
Collect automatically	G_CollectAuto	Goal
Participants use system calendar	D_ParticUseCal	DomainAssumption
Collect from system calendar	T_CollectCal	Task
Find a suitable room	G_FindRoom	Goal
Use local room	G_UseLocal	Goal
Find a local room	G_FindLocal	Goal
Get room suggestions	T_GetSuggest	Task
List available rooms	T_ListAvail	Task
Local rooms available	D_LocalAvail	DomainAssumption
Book room	G_BookRoom	Goal
Use available room	T_UseAvail	Task
Cancel less important meeting	T_CancelLess	Task
Call partner institutions	T_CallPartner	Task
Call hotels and convention centers	T_CallHotel	Task
Choose schedule	G_ChooseSched	Goal
Schedule manually	T_SchedManual	Task
Let system schedule	T_SchedSystem	Task
Manage meeting	G_ManageMeet	Goal
Cancel meeting	T_CancelMeet	Task
Confirm occurrence	T_ConfirmOcc	Task
Low cost	S_LowCost	Softgoal
Meetings cost less than €100	Q_CostLess100	QualityConstraint
Good participation	S_GoodPartic	Softgoal
A least 90% of participants attend	Q_Min90pctPart	QualityConstraint
Fast scheduling	S_FastSched	Softgoal
Schedules produced in less than a day	Q_Sched1Day	QualityConstraint

as an OCL invariant on the class `T_CharactMeet`, which, according to Table 3.2, is a subclass of `Task` (from Figure 3.2) and represents requirement *Characterize meeting*. The invariant dictates that instances of `T_CharactMeet` should never be in the `Failed` state, i.e., *Characterize meeting* should never fail.

Listing 3.1: The *AwReqs* of the Meeting Scheduler, specified in OCL_{TM}.

```
1  package meetingscheduler
2
3  -- AwReq AR1: task 'Characterize meeting' should never fail.
4  context T_CharactMeet
5    inv AR1: never(self.oclInState(Failed))
6
7  -- AwReq AR2: QC 'Meetings cost less than Euro 100' should be satisfied 75% of
       the time.
8  context Q_CostLess100
9    def: all : Set = Q_CostLess100.allInstances()
10   def: success : Set = all->select(x | x.oclInState(Succeeded))
11   inv AR2: always(success->size() / all->size() >= 0.75)
12
13 -- AwReq AR3: the success rate of goal 'Collect timetables' should not decrease
       two weeks in a row.
14 context G_CollectTime
15   def: all : Set = G_CollectTime.allInstances()
16   def: w1 : Set = all->select(x | new Date().difference(x.time, DAYS) <= 7)
17   def: w2 : Set = all->select(x | (new Date().difference(x.time, DAYS) > 7) and
         (new Date().difference(x.time, DAYS) <= 14))
18   def: w3 : Set = all->select(x | (new Date().difference(x.time, DAYS) > 14) and
         (new Date().difference(x.time, DAYS) <= 21))
19   def: success1 : Set = w1->select(x | x.oclInState(Succeeded))
20   def: success2 : Set = w2->select(x | x.oclInState(Succeeded))
21   def: success3 : Set = w3->select(x | x.oclInState(Succeeded))
22   def: rate1 : Real = success1->size() / w1->size()
23   def: rate2 : Real = success2->size() / w2->size()
24   def: rate3 : Real = success3->size() / w3->size()
25   inv AR3: never((rate1 < rate2) and (rate2 < rate3))
26
27 -- AwReq AR4: goal 'Find a suitable room' should never fail.
28 context G_FindRoom
29   inv AR4: never(self.oclInState(Failed))
30
31 -- AwReq AR5: goal 'Choose schedule' should never fail.
32 context G_ChooseSched
33   inv AR5: never(self.oclInState(Failed))
34
35 -- AwReq AR6: QC 'At least 90% of participants attend' should have a 75% success
       rate per month.
36 context Q_Min90pctPart
37   def: all : Set = Q_Min90pctPart.allInstances()
38   def: month : Set = all->select(x | new Date().difference(x.time, MONTHS) == 1)
39   def: monthSuccess : Set = month->select(x | x.oclInState(Succeeded))
40   inv AR6: always(monthSuccess->size() / month->size() >= 0.75)
41
42 -- AwReq AR7: DA 'Participants use the system calendar' should always be true.
43 context D_ParticUseCal
44   inv AR7: never(self.oclInState(Failed))
45
46 -- AwReq AR8: DA 'Local rooms available' should be false no more than once a
       week.
47 context D_LocalAvail
48   def: all : Set = D_LocalAvail.allInstances()
49   def: week : Set = all->select(x | new Date().difference(x.time, DAYS) <= 7)
50   def: weekFail : Set = week->select(x | x.oclInState(Failed))
51   inv AR8: always(weekFail.size() <= 1)
52
53 -- AwReq AR9: task 'Let system schedule' should successfully execute at least
       ten times as much as task 'Schedule manually'.
54 context T_SchedSystem
```

```
55    def: allS : Set = T_SchedSystem.allInstances()
56    def: allM : Set = T_SchedManual.allInstances()
57    def: successS : Set = allS->select(x | x.oclInState(Success))
58    def: successM : Set = allM->select(x | x.oclInState(Success))
59    inv AR9: always(successS.size() >= 10 * successM.size())
60
61  -- AwReq AR10: QC 'Schedules produced in less than a day' should have 90\%
        success rate over the past ten days, checking daily.
62  context Q_Sched1Day
63    def: all : Set = Q_Sched1Day.allInstances()
64    def: past10d : Set = all->select(x | new Date().difference(x.time, DAYS) <=
          10)
65    def: success10d : Set = past10d->select(x | x.oclInState(Succeeded))
66    -- @daily
67    inv AR10: always(success10d->size() / past10d->size() >= 0.90)
68
69  -- AwReq AR11: goal 'Manage meeting' should be satisfied within one hour of the
        time set by the meeting's schedule.
70  context G_ManageMeet
71    def: meet : Meeting = self.argument("meeting")
72    inv AR11: eventually(self.oclInState(Succeeded) and (self.time.difference(meet
          .startTime, MINUTES) <= 60))
73
74  -- AwReq AR12: task 'Confirm occurrence' should be decided within five minutes.
75  context T_ConfirmOcc
76    inv AR12: eventually(not self.oclInState(Undecided) and (self.time.difference(
          self.startTime, MINUTES) <= 5))
77
78  -- AwReq AR13: AwReq 'AR7' should succeed 80% of the times.
79  context AR7
80    def: all : Set = AR7.allInstances()
81    def: success : Set = all->select(x | x.oclInState(Succeeded))
82    inv AR13: always(success->size() / all->size() >= 0.8)
83
84  -- AwReq AR14: the monthly success rate of AwReq 'AR6' should not decrease twice
        in a row.
85  context AR14
86    def: all : Set = AR6.allInstances()
87    def: m1 : Set = all->select(x | new Date().difference(x.time, MONTHS) == 1)
88    def: m2 : Set = all->select(x | new Date().difference(x.time, MONTHS) == 2)
89    def: m3 : Set = all->select(x | new Date().difference(x.time, MONTHS) == 3)
90    def: success1 : Set = m1->select(x | x.oclInState(Succeeded))
91    def: success2 : Set = m2->select(x | x.oclInState(Succeeded))
92    def: success3 : Set = m3->select(x | x.oclInState(Succeeded))
93    def: rate1 : Real = success1->size() / m1->size()
94    def: rate2 : Real = success2->size() / m2->size()
95    def: rate3 : Real = success3->size() / m3->size()
96    inv AR14: never((rate1 > rate2) and (rate2 > rate3))
```

Aggregate *AwReqs* place constraints over a collection of instances. In *AR2*, for example, all instances of `Q_CostLess100` are retrieved in a set named `all`, then we use the `select()` operation to separate the subset of the instances that succeeded and, finally, we compare the sizes of these two sets in order to assert that 75% of the instances are successful at all times (`always`).

In some aggregate *AwReqs*, instances have to be aggregated in a specific period of time. Since OCL does not provide a type and operations for dates, we follow the syntax exemplified by Robinson [2008] in the paper in which he proposed OCL$_{TM}$. This is illustrated by *AR6*, which uses the `select()` operation to obtain the subset of all instances of `Q_Min90pctPart` whose `time` attribute is exactly one month different than the current date (provided by `new Date()`), thus aggregating instances from last month. When the

verification frequency of an *AwReq* is specified, we add an CRON[4]-style annotation as a comment before the invariant. For example, *AR10*'s invariant was annotated with `@daily`, specifying the *AwReq* should be checked every 24 hours.

Trend *AwReqs* are similar, but a bit more complicated as we must separate the requirements instances into different time periods. For *AR3*, the `select()` operation was used to create sets with the instances of `G_CollectTime` for the past three weeks to compare the rate of success over time.

Delta *AwReqs* specify invariants over single instances of the requirements. *AR11* singles out the instance of the `Meeting` domain class that is related to a specific `G_ManageMeet` instance and its invariant states that the instance that represents the goal should be eventually satisfied and, moreover, that should happen within one hour of the meeting's start time. As stated before, our framework does not yet support integration with domain models and, for this reason, we do not prescribe any specific syntax for them.

Finally, *AR12* shows how to specify the example in which we do not talk specifically about success or failure of a requirement, but its change of state: eventually instances of `T_ConfirmOcc` should not be in the `Undecided` state and the difference between their start and end times should be at most five minutes.

3.1.3 Patterns and graphical representation

It can be seen from the illustrations in the previous section that specifying *AwReqs* is not a trivial task. For this reason we propose *AwReq* patterns to facilitate their elicitation and analysis and a graphical representation that allows us to include them in the goal model, improving communication among system analysts and designers.

Many *AwReqs* have similar structure, such as "something must succeed so many times". By defining patterns for *AwReqs* we create a common vocabulary for analysts. Furthermore, patterns are used in the graphical representation of *AwReqs* in the goal model and code generation tools could be provided to automatically write the *AwReq* in the language of choice based on the pattern. In Chapter 6, we provide OCL_{TM} idioms for this kind of code generation. We expect that the majority (if not all) *AwReqs* fall into these patterns, so their use can relieve requirements engineers from most of the specification effort.

Table 3.3 contains a list of patterns that we have identified so far in our research on this topic. This list is by no means exhaustive and each organization is free to define their own patterns (with their own names and meanings). Furthermore, it is important to note that when requirements engineers create patterns, they are responsible for their

[4]See `http://en.wikipedia.org/wiki/CRON_expression`.

Table 3.3: A non-exhaustive list of *AwReq* patterns.

Pattern	**Meaning**
`NeverFail(R)`	Requirement `R` should never fail. Analogous patterns `AlwaysSucceed`, `NeverCanceled`, etc.
`SuccessRate(R, r, t)`	`R` should have at least success rate `r` over time `t`. Analogous patterns `FailureRate`, `CancelationRate`, etc.
`SuccessRateExecutions (R, r, n)`	`R` should have at least success rate `r` over the latest `n` executions. Analogous patterns `FailureRateExecutions`, `CancelationRateExecutions`, etc.
`MaxFailure(R, x, t)`	`R` should fail at most `x` times over time `t`. Analogous patterns `MinFailure`, `MinSuccess`, etc.
`ComparableSuccess(R, S, x, t)`	`R` should succeed at least `x` times more than `S` over time `t`. Analogous patterns `ComparableFailure`, `ComparableCancelations`, etc.
`TrendDecrease(R, t, x)`	The success rate of `R` should not decrease `x` times consecutively considering periods of time specified by `t`. Analogous pattern `TrendIncrease`.
`ComparableDelta(R, S, p, x)`	The difference between the value of attribute `p` in requirements `R` and `S` should not be greater than `x`.
`StateDelta(R, s1, s2, t)`	`R` should transition from state `s1` to state `s2` in less time than what is specified in `t`.
P_1 `and` / `or` P_2; `not` P	Conjunction, disjunction and negation of patterns.

consistency and correctness and, unfortunately, our approach does not provide any tool to help in this task.

Given these patterns, the *AwReqs* of the Meeting Scheduler shown back in Table 3.1 can now be more concisely documented and communicated, as shown in the right-most column of Table 3.4. For values representing periods of time, abbreviated amounts of time like in OCL_{TM} timeouts [Robinson, 2008] were used.

Given that *AwReqs* can be shortened by a pattern we propose to represent them graphically in the goal model along with other elements such as goals, tasks, softgoals, etc. When producing requirement specification documents, analysts can choose between using this graphical representation or documenting *AwReqs* in tables, such as Table 3.4. On one side, the graphical representation might overload the model, on the other side, they provide everything in one place, which is more practical. Figure 3.3 shows the

Table 3.4: Example *AwReqs* described in natural language and represented with *AwReq* patterns.

Id	Description	Pattern
AR1	Task *Characterize meeting* should never fail.	`NeverFail(T_CharactMeet)`
AR2	Quality constraint *Meetings cost less than €100* should be satisfied 75% of the time.	`SuccessRate(Q_CostLess100, 75%)`
AR3	The success rate of goal *Collect timetables* should not decrease two weeks in a row.	`not TrendDecrease(G_CollectTime, 7d, 2)`
AR4	Goal *Find a suitable room* should never fail.	`NeverFail(G_FindRoom)`
AR5	Goal *Choose schedule* should never fail.	`NeverFail(G_ChooseSched)`
AR6	Quality constraint *At least 90% of participants attend* should have a 75% success rate per month.	`SuccessRate(Q_Min90pctPart, 90%, 1M)`
AR7	Domain assumption *Participants use the system calendar* should always be true.	`NeverFail(D_ParticUseCal)`
AR8	Domain assumption *Local rooms available* should be false no more than once a week.	`MaxFailure(D_LocalAvail, 1, 7d)`
AR9	Task *Let system schedule* should successfully execute at least ten times as much as task *Schedule manually.*	`ComparableSuccess(T_SchedSystem, T_SchedManual, 10)`
AR10	Quality constraint *Schedules produced in less than a day* should have 90% success rate over the past ten days, checking daily.	`@daily SuccessRate(Q_Sched1Day, 90%, 10d)`
AR11	Goal *Manage meeting* should be satisfied within one hour of the time set by the meeting's schedule.	–
AR12	Task *Confirm occurrence* should be decided within five minutes.	`StateDelta(T_ConfirmOcc, Undecided, *, 5m)`
AR13	*AwReq AR7* should succeed 80% of the times.	`SuccessRate(AR7, 80%)`
AR14	The monthly success rate of *AwReq AR6* should not decrease twice in a row.	`not TrendDecrease(AR6, 30d, 2)`

Meeting Scheduler's goal model with added *AwReqs*.

AwReqs are represented by thick circles with arrows pointing to the target to which they refer and the *AwReq* pattern besides it. The first parameter of the pattern is omitted, as the *AwReq* is pointing to it. In case an *AwReq* does not fit a pattern (e.g., *AR11*), the analyst should write its identifier and document its specification elsewhere (e.g., Listing 3.1). In Figure 3.3, the *AwReqs*' identifiers are provided not only for *AR11*, but for all *AwReqs* in order to to facilitate referencing the model.

An important remark here is that, in this thesis, we have not applied any methodology for the construction of visual notations (e.g., [Moody, 2009]). Instead, we used simple analogies to decide on basic geometric figures that could graphically represent the newly proposed concepts. In the case of *AwReqs*, the circle was chosen for it has a similar format to an eye, and *AwReqs* define the requirements that should be monitored — in other words, "what to look for" — at runtime.

3.2 Evolution Requirements

It is often the case that the requirements elicited from stakeholders for a system-to-be are not carved in stone, never to change during the system's lifetime. Rather, stakeholders will often hedge with statements such as "If requirement R fails more than N times in a week, relax it to R-", or "If we find that we are fulfilling our target (requirement S), let's strengthen S by replacing it with $S+$", or even "Requirement Q no longer applies after January 1st, 2013".

As explained back in Section 1.2.4 (p. 10), these are all requirements in the sense that they come from stakeholders and describe desirable properties of the system-to-be. They are special requirements, however, in the sense that their operationalization consists of changing other requirements, as suggested by the examples above.

A requirements model defines a space of system behaviors, where each behavior fulfills system objectives. When reconfiguration is performed, like in some of the approaches summarized in Section 2.2.3 (cf. *Reconfiguration approaches*, in p. 55), a new behavior is selected from this space of alternatives. In this chapter, however, we concentrate on requirements that change that space, thereby defining a changed set of system behaviors (our proposals for adaptation through reconfiguration are presented in the following chapter). Such evolutions allow the system to utilize new alternative behaviors.

We call such requirements *Evolution Requirements* (a.k.a. *EvoReqs*) since they prescribe desired evolutions for other requirements. At runtime, *EvoReqs* have an effect on the running components of the system with the purpose of meeting stakeholder directives. *EvoReqs* allow us to not only specify what other requirements need to change, but also

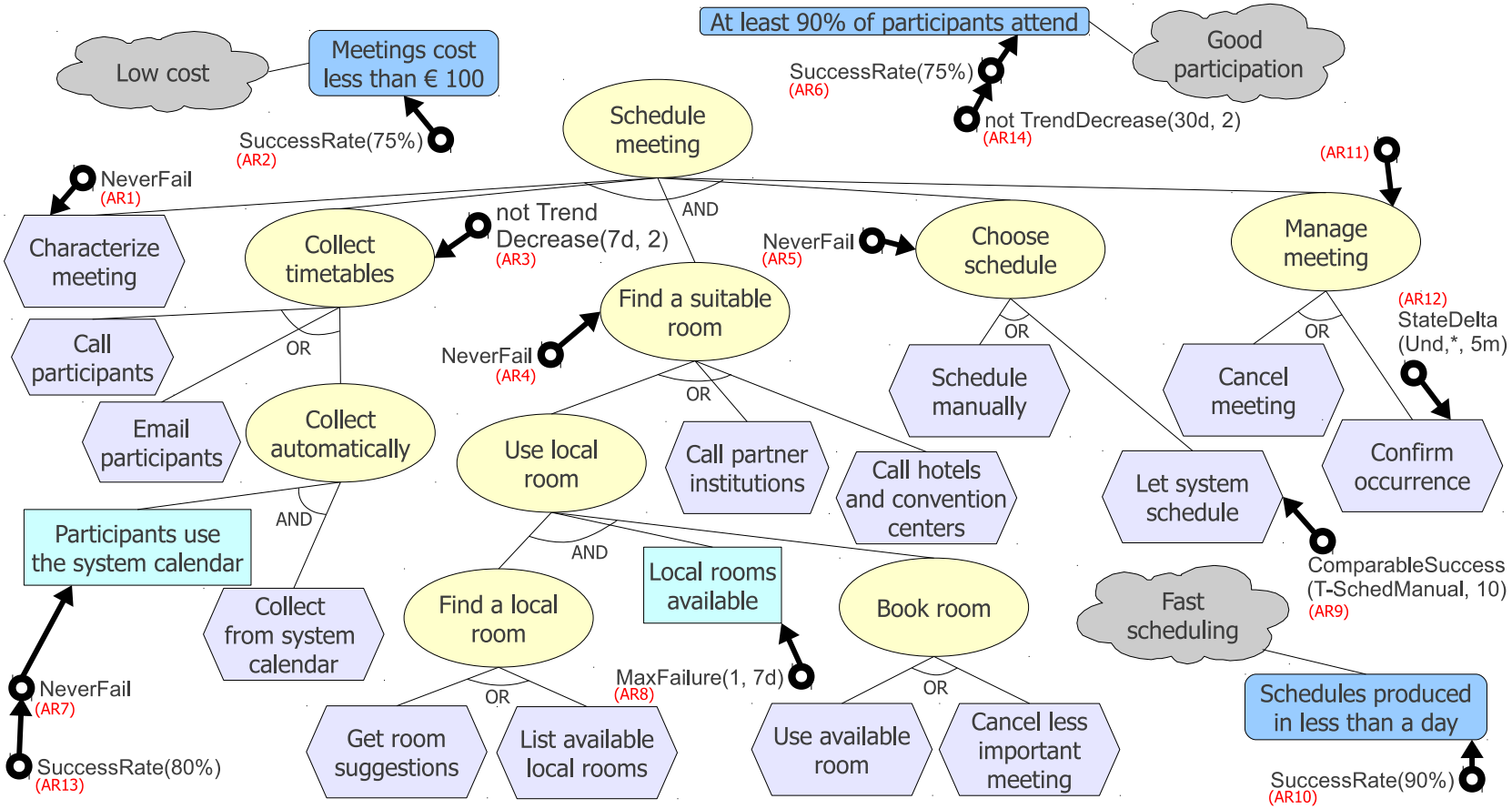


Figure 3.3: GORE-based specification for the Meeting Scheduler, with *AwReqs* added using their graphical representation.

when other strategies — such as "retry after some time" or "relax the requirement" — should be used.

Later, in Chapter 6, we will detail a framework that uses Event-Condition-Action (ECA) rules to operationalize this kind of requirement. In this chapter, we will focus on how to model *EvoReqs* in terms of the three ECA components: the event component is, as before, an *AwReq* failure; the condition is elicited from stakeholders in terms of when to use one or other particular strategy such as, for instance, "this is applicable only once", "apply this only between midnight and 6 AM", and so forth.

Finally, the action component of the ECA rule consists of a sequence of primitive operations on a goal model (that evolve it according to stakeholder wishes). Each operation effects a primitive change on the model, e.g., removes/adds a goal at the class or instance level, changes the state of a goal instance, or undoes the effects of all executed actions for an aborted execution. Furthermore, such operations can be combined using patterns in order to compose macro-level evolution strategies, such as *Retry* and *Relax*.

In the following sub-sections, we characterize *EvoReqs* and present some patterns that can facilitate their elicitation and representation. Moreover, we show how reconfiguration approaches could be integrated in our models by having primitive operations that execute reconfiguration algorithms and apply the new configuration in the target system. Later, in Chapter 4, we integrate our own reconfiguration algorithms and provide the complete specification for the adaptation capabilities of the Meeting Scheduler, in order to illustrate these new modeling concepts.

3.2.1 Characterization

Evolution requirements specify changes to other requirements when certain conditions apply. For instance, suppose the stakeholders provide the following requirements:

- If the meeting organizer fails to *Characterize meeting* (*AR1*), she should **retry** after a few seconds;
- If there is a negative trend on the success rate of *Collect timetables* for two consecutive weeks (*AR3*), we can tolerate this at most once per trimester, **relaxing** the constraint to three weeks in a row;
- If local rooms are often unavailable (`AR8`), the meeting scheduler software cannot autonomously create new rooms (i.e., increase `RfM`). This task should be **delegated** to the management;
- If we realize that the domain assumption *Participants use the system calendar* is not true (*AR7*), **replace it** with a task that will enforce the usage of the system

calendar.

We propose to represent these requirements by means of sequence of operations over goal model elements, in a way that can be exploited at runtime by an adaptation framework (cf. Section 6) which, acting like a controller in a control system, sends adaptation instructions to the target system. We call them *Evolution Requirements* (*EvoReqs*).

EvoReqs and *AwReqs* (cf. Section 3.1) complement one another, allowing analysts to specify the requirements for a feedback loop that operationalizes adaptation at runtime: *AwReqs* indicate the situations that require adaptation and *EvoReqs* prescribe what to do in these situations. It is important to note, however, that *EvoReqs* are not the only way to adapt to *AwReq* failures. Analogously, *AwReq* failures are not the only event that can trigger *EvoReqs* (the framework proposed in this thesis could be adapted to respond to, e.g., scheduled events).

EvoReqs, thus, are specified as a sequence of primitive operations which have an effect on the target system (TS) and/or on the adaptation framework (AF) itself, effectively telling them how to change (or, using a more evolutionary term, "mutate") the requirements model in order to adapt. The existing operations and their respective effects are shown in Table 3.5 and could also be extended if necessary.

As can be seen in the table, adaptation instructions have arguments which can refer to, among other things, system actors (`A`), requirements classes (upper-case `R`) or instances (lower-case `r`) and system parameters (`p`) and their values (`v`). Actors can be provided by any diagram that models external entities that interact with the system, e.g., $i^\star$ Strategic Dependency models [Yu et al., 2011].

Requirements classes/instances are provided by the specification of *AwReqs*: as we have seen in Section 3.1, each element of the goal model is represented as a UML class, extending the super-classes shown in Figure 3.2 (p. 71). Run-time instances of these elements (such as the various meetings being scheduled) are then represented as objects that instantiate these classes.

Finally, parameters are elicited during system identification, as will be explained later, in chapters 4 and 5. Instructions `apply-config` and `find-config` also refer to configurations (`C`) and algorithms (`algo`) — we will come back to these concepts in Section 3.2.3.

Listing 3.2 shows the specification of one of the examples presented earlier in this section: retry a requirement when it fails (in the example, the qualitative value *few* has been replaced by a quantitative value of 5 seconds). Here, `r` represents an instance of task *Characterize meeting*, referred to by the instance of *AwReq AR1* that failed. The framework then creates another instance of the task, tells the target system to copy the data from the execution session of the failed task to the one of the new task, to terminate the failing components and rollback any partial changes made by them. After $5s$, the

Table 3.5: Evolution requirements operations and their effect.

Instruction	**Effect**
`abort(ar)`	TS should "fail gracefully", which could range from just showing an error message to shutting the entire system down, depending on the system and the *AwReq* `ar` that failed.
`apply-config(C, L)`	TS should change from its current configuration to the specified configuration `C`. `L` indicates if the change should occur at the class level (for future executions) and/or at the instance level (current execution).
`change-param([R|r], p, v)`	TS should change the parameter `p` to the value `v` for either all future executions of requirement `R` or the current requirement instance `r`.
`copy-data(r, r')`	TS should copy the data associated with performative requirement instance `r` (e.g., data provided by the user) to instance `r'`.
`disable(R)`, `suspend(r)`	TS should stop trying to satisfy requirement instance `r` in the current execution, or requirement `R` from now on. If `r` (or `R`) is an *AwReq*, AF should stop evaluating it.
`enable(R)`, `resume(r)`	TS should resume trying to satisfy requirement instance `r` in the current execution, or requirement `R` from now on. If `r` (or `R`) is an *AwReq*, AF should resume evaluating it.
`find-config(algo, ar)`	AF should execute algorithm `algo` to find a new configuration for the target system with the purpose of reconfiguring it. Other than the *AwReq* instance `ar` that failed, AF should provide to this algorithm the system's current configuration and the system's requirements model.
`initiate(r)`	TS should initialize the components related to `r` and start pursuing the satisfaction of this requirement instance. If `r` is an *AwReq* instance, AF should immediately evaluate it.
`new-instance(R)`	AF should create a new instance of requirement `R`.
`rollback(r)`	TS should undo any partial changes that might have been effected while the satisfaction of performative requirement instance `r` was being pursued and which would leave the system in an inconsistent state, as in, e.g., Sagas [Garcia-Molina and Salem, 1987].
`send-warning(A, ar)`	TS should warn actor `A` (human or system) about the failure of *AwReq* instance `ar`
`terminate(r)`	TS should terminate any component related to `r` and stop pursuing the satisfaction of this requirement instance. If `r` is an *AwReq* instance, AF should no longer consider its evaluation.
`wait(t)`	AF should wait for the amount of time `t` before continuing with the next operation. TS is also informed of the wait in case changes in the user interface are in order during the waiting time.
`wait-for-fix(ar)`	TS should wait for a certain condition that indicates that the problem causing the failure of *AwReq* `ar` has been fixed.

framework finally instructs the target system to initiate the new task, thus accomplishing "retry after a few seconds".

Listing 3.2: *EvoReq* "Retry *Characterize meeting* after 5 seconds"

```
1 r' = new-instance(T_CaractMeet);
2 copy-data(r, r');
3 terminate(r);
4 rollback(r);
5 wait(5s);
6 initiate(r');
```

Although evolution operations are generic, their effects on the target system are application-specific. For example, instructing the system to try a requirement again could mean, depending on the system and the requirement, retrying some operations autonomously or showing a message to the user explaining that she should repeat the actions she has just performed. Therefore, in order to be able to carry out these operations, the target system is supposed to implement an *Evolution API* that receives all operations of Table 3.5, for each requirement in the system's model. Obviously, as with any other requirement in a specification, each operation–requirement pair can be implemented on an as-needed basis.

Revisiting the previous example, `copy-data` should tell the Meeting Scheduler to copy the data related to the task that failed (e.g., information on the meeting that has already been filled in the system) to a new user session, `terminate` closes the screen that was being used by the meeting organizer to characterize the meeting, `rollback` deletes any partial changes that might have been saved, `wait` shows a message asking the user to wait for $5s$ and, finally, `initiate` should open a new screen associated with the new user session so the meeting organizer can try again. All this behavior is specific to the Meeting Scheduler and the task at hand and the way it will be implemented depends highly on the technologies chosen during its architectural design.

3.2.2 Adaptation Strategies as Patterns

The operations of Table 3.5 allow us to describe different *adaptation strategies* in response to *AwReqs* failures using *EvoReqs*. However, many *EvoReqs* might have similar structures, such as "wait t seconds and try again, with or without copying data". Therefore, to facilitate their elicitation and modeling, we propose the definition of patterns that represent common adaptation strategies. Table 3.6 shows the specification for some *EvoReq* patterns.

A strategy is defined by a name, a list of arguments that it accepts (with optional default values) and an algorithm (composed of Java™-style pseudo-code and evolution operations) to be carried out when the strategy is selected. Strategies are usually associated to failures of *AwReqs* and, therefore, we can also refer to the instance of the

Table 3.6: Some *EvoReq* patterns and their specifications based on *EvoReq* operations.

```
Abort() {
  abort(awreq);
}
```

```
Delegate(a : Actor) {
  send-warning(a, awreq);
  wait-for-fix(awreq);
}
```

```
RelaxDisableChild(r : Requirement = awreq.target; level : Level = INSTANCE;
    child : Requirement) {
  if ((level == CLASS) || (level == BOTH)) disable(child.class);
  if ((level == INSTANCE) || (level == BOTH)) {
    suspend(r);
    terminate(child);
    if (child.class = PerformativeRequirement) rollback(child);
    suspend(child);
    resume(r);
  }
}
```

```
Replace(r : Requirement = awreq.target; copy : boolean = true; level : Level =
    INSTANCE; r' : Requirement) {
  R = r.class;
  R' = r'.class;
  if ((level == CLASS) || (level == BOTH)) {
    disable(R);
    enable(R');
  }
  if ((level == INSTANCE) || (level == BOTH)) {
    if (R = PerformativeRequirement) && (R' = PerformativeRequirement)
      && (copy) copy-data(r, r');
    terminate(r);
    if (R = PerformativeRequirement) rollback(r);
    suspend(r);
    initiate(r');
  }
}
```

```
Retry(copy: boolean = true; time: long) {
  r = awreq.target; R = r.class;
  r' = new-instance(R);
  if (copy) copy-data(r, r');
  terminate(r); rollback(r);
  wait(time);
  initiate(r');
}
```

```
StrengthenEnableChild(r : Requirement = awreq.target; level : Level = INSTANCE;
    child : Requirement) {
  if ((level == CLASS) || (level == BOTH)) enable(child.class);
  if ((level == INSTANCE) || (level == BOTH)) {
    suspend(r);
    resume(child);
    initiate(child);
    resume(r);
  }
}
```

```
Warning(a : Actor) {
  send-warning(a, awreq);
}
```

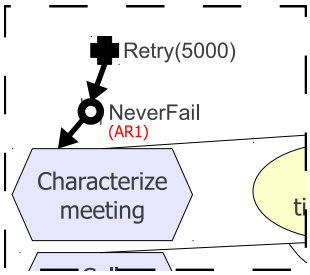


Figure 3.4: Graphical representation of an adaptation strategy in response to an *AwReq* failure.

AwReq that failed using the keyword `awreq` in the pseudo-code. Given the `Retry` strategy in Table 3.6, and assuming that time is represented in milliseconds, the example from Section 3.2.1 could be more concisely expressed as `Retry(5000)`.

It is important to note, however, that the list in Table 3.6 is not intended to be exhaustive and new strategies can be created as needed. For instance, one could take inspiration from the design patterns for adaptation cataloged by Ramirez and Cheng [2010]. After strategies have been elicited and represented as patterns, they can be associated with *AwReqs* and added to the requirements specification.

The use of patterns also allow us to add adaptation strategies to the goal model, as shown in Figure 3.4. This portion of the Meeting Scheduler's model shows the `Retry(5000)` pattern associated with failures of *AwReq AR1*. The analogy behind the choice of the cross as graphical representation comes from the red cross symbol usually associated with hospitals and emergency rooms, which can "fix ill people" the same way as adaptation strategies may "fix the system".

3.2.3 Reconfiguration as an adaptation strategy

As introduced back in Section 1.2.4 (p. 10), our approach proposes two modes of adaptation: evolution and reconfiguration. *EvoReqs* correspond to the former, whereas the latter could be provided by any of the *Reconfiguration approaches* cited in page 55. Later, in Chapter 4, we propose our own reconfiguration approach.

In any case, the decision to use reconfiguration (and which reconfiguration algorithm to use) should belong to the stakeholders and domain experts. In other words, "If requirement *R* fails, reconfigure the system (using *this* or *that* algorithm) in order to improve its success rate" is also a stakeholder requirement. Although not an evolution requirement, as reconfiguration does not change the requirements themselves, we would like to provide a unified framework to represent the requirements for system adaptation.

Therefore, in order to unify both specifications, we consider *reconfiguration* a type of *adaptation strategy*. *EvoReqs* can, thus, be used to specify that stakeholders would like

to use reconfiguration, in one of two ways:

1. If stakeholders wish to apply a specific reconfiguration for a given failure, instructions like `change-param`, `enable/disable` and `initiate/terminate` can be used to describe the precise changes in requirements at class and/or instance level;

2. Instead, if there is no specific way to reconfigure, a reconfiguration algorithm that is able to compare the different alternatives should be executed using the `find-config` instruction, after which `apply-config` is called to inform the target system about the new configuration.

Listing 3.3 shows the pattern that describes the adaptation strategy of option 2. The strategy receives as arguments an algorithm to find the new configuration, the *AwReq* that failed and thus triggered the strategy and the level at which the changes should be applied: class (future executions), instance (current execution) or both. The adaptation framework executes the given reconfiguration algorithm, which returns a new system configuration. Then, this new configuration is applied to the target system at the specified level.

Listing 3.3: Reconfiguration as an adaptation strategy.

```
1 Reconfigure(algo: FindConfigAlgorithm, ar: AwReq, level: Level = INSTANCE) {
2   C' = find-config(algo, ar)
3   apply-config(C', level)
4 }
```

One important thing to note is that different reconfiguration algorithms may require different information from the model, which should be provided accordingly. In the next chapter, we discuss the kind of information required by our reconfiguration algorithms and later in Chapter 5 we propose a systematic process to elicit such information.

After unifying reconfiguration and evolution into a single way of specifying adaptation strategies, we can provide the final specification for the adaptation requirements of the Meeting Scheduler scenario. The adaptation strategies associated with each *AwReq* failure are shown in Table 3.7. Details pertaining our reconfiguration algorithms will be provided at the end of Chapter 4.

Note that, in this table, adaptation strategies are enumerated to indicate the order in which they should be applied. To complete this specification, one should associate also applicability conditions to each strategy, but this will be explained later, in Chapter 6. Moreover, we have conducted experiments with a larger system than the Meeting Scheduler, the results of which will be presented in Chapter 7.

Table 3.7: Adaptation strategies elicited for the Meeting Scheduler.

AwReq	***AwReq* pattern**	**Adaptation strategies**
AR1	`NeverFail(T_CharactMeet)`	1. *Retry(5000)* 2. *Reconfigure()*
AR2	`SuccessRate(Q_CostLess100, 75%)`	1. *Reconfigure()*
AR3	`not TrendDecrease(G_CollectTime, 7d, 2)`	1. *Replace(AR3, CLASS, AR3_3weeks)* 2. *Warning(IT_Staff)* 3. *Reconfigure()*
AR4	`NeverFail(G_FindRoom)`	1. *Reconfigure()*
AR5	`NeverFail(G_ChooseSched)`	1. *Reconfigure()*
AR6	`SuccessRate(Q_Min90pctPart, 90%, 1M)`	1. *Reconfigure()*
AR7	`NeverFail(D_ParticUseCal)`	1. *Reconfigure()* 2. *Replace(CLASS, T_EnforceUseOfCalendar)*
AR8	`MaxFailure(D_LocalAvail, 1, 7d)`	1. *Delegate(Management)* 2. *Reconfigure()*
AR9	`ComparableSuccess( T_SchedSystem, T_SchedManual, 10)`	1. *Reconfigure()*
AR10	`@daily SuccessRate(Q_Sched1Day, 90%, 10d)`	1. *Reconfigure()*
AR11	—	1. *Warning(Secretary)*
AR12	`StateDelta(T_ConfirmOcc, Undecided, *, 5m)`	1. *Warning(Secretary)*
AR13	`SuccessRate(AR7, 80%)`	1. *Reconfigure()*
AR14	`not TrendDecrease(AR6, 30d, 2)`	1. *Reconfigure()*

3.3 Chapter summary

In this chapter, we have presented the first contribution of this thesis: new classes of requirements — *Awareness Requirements* (*AwReqs*, § 3.1) and *Evolution Requirements* (*EvoReqs*, § 3.2) — that represent, respectively, the requirements for the monitoring and adaptation components of the feedback loop that operationalizes adaptation, thus promoting this loop to a first-class citizen in requirement models.

More specifically, *AwReqs* are characterized as requirements that impose constraints on the success or failures of other requirements, being divided in three categories: aggregate, delta or trend *AwReqs* (§ 3.1.1). In our approach, they are specified using an extension of the OCL language called OCL_{TM} (OCL with Temporal Message logic), referring to elements of the goal model represented as UML classes which extend superclasses from a base model (§ 3.1.2). To make modeling easier, *AwReqs* can be represented as patterns such as "never fail" or "x% success rate" and even added to the goal model using a proposed graphical representation (§ 3.1.3).

In their turns, *EvoReqs* are characterized as requirements that prescribe desired evolutions for other requirements, being represented by a sequence of basic evolution operations over elements of the goal model, each of which representing a particular effect on the adaptation framework or the target system (§ 3.2.1). As with *AwReqs*, *EvoReqs* can also be represented more concisely as patterns which are called *adaptation strategies* (§ 3.2.2). One such strategy is reconfiguration: given a failure and a reconfiguration algorithm, execute the algorithm to find a new system configuration in order to adapt (§ 3.2.3). Later in this thesis, we propose a family of reconfiguration algorithms that can be used in combination with this strategy.

The Meeting Scheduler, introduced in Chapter 2, was used throughout the chapter to illustrate all the aforementioned new concepts. A goal model of this example system with added *AwReqs* can be seen in Figure 3.3 (p. 79) and a partial specification of the scheduler's monitoring and adaptation requirements are listed in Table 3.7 (p. 87). This table will be completed later, after our proposal for reconfiguration algorithms is properly presented.

Chapter 4

Qualitative Adaptation Mechanisms

Adapt or perish, now as ever, is Nature's inexorable imperative.

H. G. Wells

As the previous chapter has shown, *AwReqs* can be used to determine when requirements are not being satisfied (*requirements divergence*), much the same way a control system calculates the control error (cf. Section 2.1.5, p. 39), i.e., the discrepancy between the reference input (*system requirements*) and the measured output (*indicators of requirements convergence*, or simply *indicators*). The next step, then, is to determine the control input based on this discrepancy, i.e., determine what could be done to adapt the target system in order to ultimately satisfy the requirements.

In Control Theory (e.g., [Hellerstein et al., 2004; Doyle et al., 1992]), the first step towards accomplishing this is an activity called *System Identification*, which is the process of determining the equations that govern the dynamic behavior of a system. This activity is concerned with: (a) the identification of system parameters that, when manipulated, have an effect on the measured output; and (b) the understanding of the nature of this effect. Afterwards, these equations can guide the choice of the best way to adapt to different circumstances. For example, in a control system in which the room temperature is the measured output, turning on the air conditioner lowers the temperature, whereas using the furnace raises it. If the heating/cooling systems offer different levels of power, there is also a relation between such power level and the rate in which the temperature in the room changes.

White box models describe a system from first principles, e.g., a model for a physical process that consists of Newton equations that describe the relations between parameters and outputs. In most cases, however, such models are overly complicated or even impossible to obtain due to the complex nature of many systems and processes (natural or artificial). A much more common approach is therefore to start from partial knowledge

of the behavior of the system and its external influences (inputs), and try to determine a mathematical relation between inputs and outputs without going into the details of what is actually happening inside the system. Two types of models are built using this approach:

- *Gray box models*: although the peculiarities of system internals are not entirely known, a certain model based on both insight into the system and experimental data is constructed. This model, however, comes with a number of free parameters (control variables) which can be estimated using system identification. Thus, parameter estimation is an important activity here;
- *Black box models*: no prior model is available here, so everything has to be constructed from scratch, through observation and experimentation. Most system identification algorithms are of this type.

Our proposal is to employ this control-theoretic framework for the design of adaptive software systems, adopting a GORE perspective, which means to assume that a goal-based requirements model is available for the system. At the requirements level, the system is not yet implemented and its behavior is not completely known. With this incomplete information, we are unable to fully identify how system configuration parameters affect outputs. Thus, quantitative approaches cannot be applied.

Therefore, we base our approach on ideas from Qualitative Reasoning (cf. Section 2.1.7, p. 45, also [Kuipers, 1989; Forbus, 2004]) and propose new modeling constructs that identify target indicators and system configuration parameters as well as qualitative relations between these parameters and measured indicators. The proposed constructs are both qualitative and flexible in the sense that they can accommodate multiple levels of precision in specifications depending on available information.

According to our proposal, the output of system identification for a software system is an extended and parametrized requirements model. Each assignment of parameter values represents a different behavior (configuration) that the system might adapt to fulfill its requirements. Some of the parameters — called *variation points* — come directly from the model. Take, for instance, the requirements for the Meeting Scheduler with added variability shown back in Figure 2.6 (p. 37). For this system to collect timetables from from all participants when a meeting is scheduled, there is a choice of collecting these directly from meeting participants (via telephone or email) or from a central repository of timetables (the system calendar).

Additionally, system behaviors are also determined by a set of *control variables* that influence system execution, its success rate, performance, or quality of service. For instance, again in the Meeting Scheduler, the *Collect timetables* goal is influenced by a

parameter *From how Many* (FhM) that determines from what percentage of the participants we need to collect timetables before the goal is deemed to have been fulfilled. If we need to collect from all, i.e., $FhM = 100\%$, then the success rate for the goal may be low and its completion time may be high, compared to a $FhM = 80\%$ setting.

In this chapter, we propose a language for modeling qualitative information on the relation between system parameters and indicators. However, the process through which these relations are elicited — i.e., *System Identification* — is covered later, in Chapter 5. Below, Section 4.1 provides the means of representing *system parameters* and how they affect the monitored indicators. Then, Section 4.2 complements this previous specification with information that allows the feedback loop to perform *qualitative adaptation* through system reconfiguration.

We continue to illustrate our modeling concepts with examples from the Meeting Scheduler, building on the model presented back in Figure 3.3 (p. 79).

4.1 Indicator/parameter relations

As previously introduced, we propose a language to model qualitative relations between configuration parameters and measured outputs of the system, in order to allow the system to reconfigure itself at runtime. Before specifying these relations, however, we have to identify the subjects of the relation: on one side, *variation points* and *control variables* (collectively called *parameters*), and on the other side, *indicators* (of requirements convergence).

4.1.1 System parameters and indicators

According to Semmak et al. [2008], the concept of *Variation points* (VPs) comes from the field of feature modeling [Griss et al., 1998]. In GORE-based specifications, they are represented as OR-refinements, which might already be present in the goal model if variability was a concern when the system's requirements were elicited (cf. Section 2.1.4, p. 36).

Therefore, our proposal simply adds labels to VPs in the goal model, in order to refer to them when modeling qualitative relations, as we will see later in Section 4.1.3. Figure 4.1 shows five VPs identified in the Meeting Scheduler, labeled *VP1*–*VP5*. Note that not all OR-refinements are necessarily variation points: the refinement of goal *Manage meeting* just shows two possible outcomes for a scheduled meeting: it is either canceled by the organizer or confirmed by a secretary.

Moreover, we propose *Control Variables* (CVs), which represent another powerful mechanism for system reconfiguration. CVs are part of the system input and can be

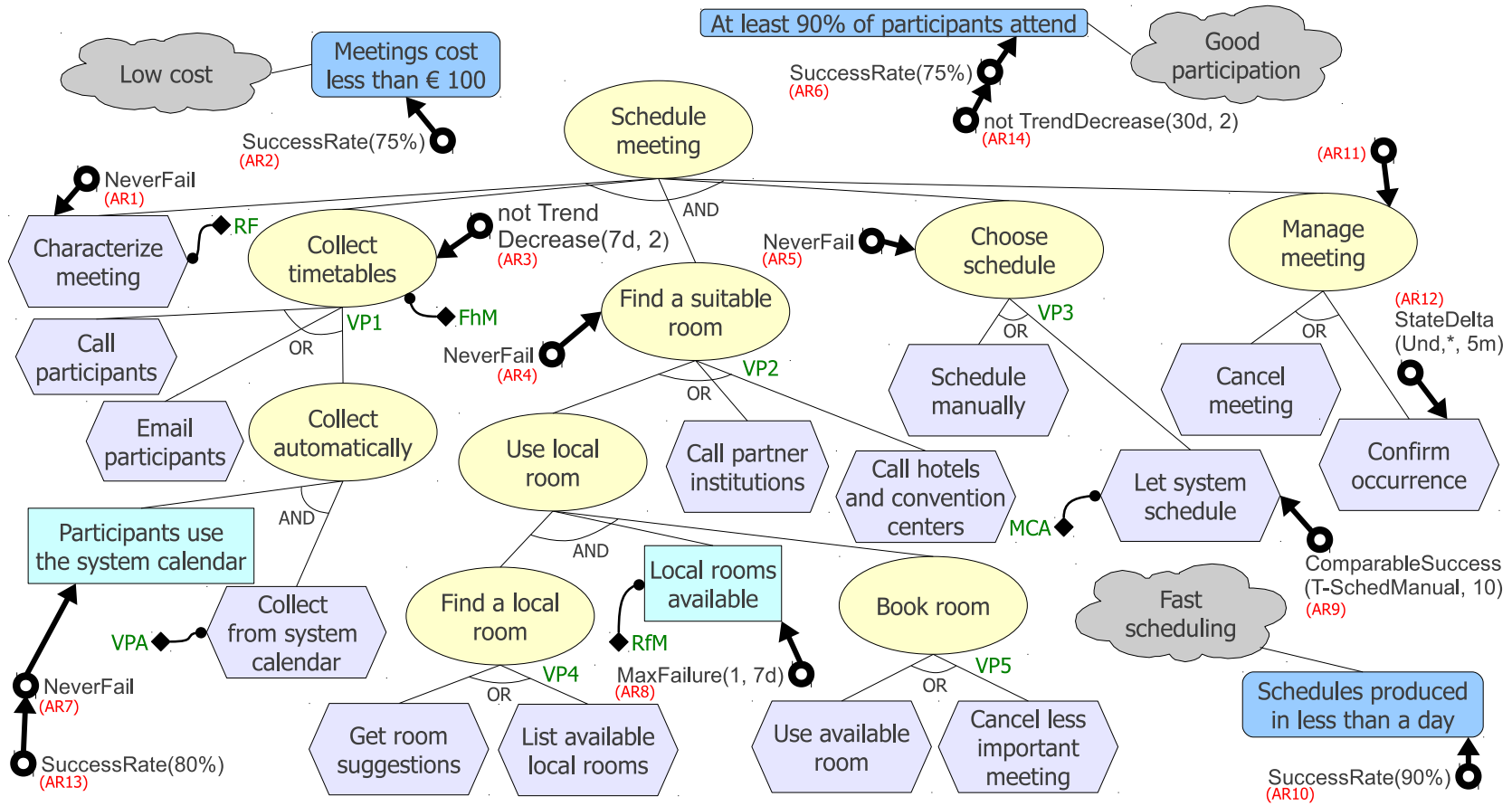


Figure 4.1: The specification for the Meeting Scheduler, augmented with variation points and control variables.

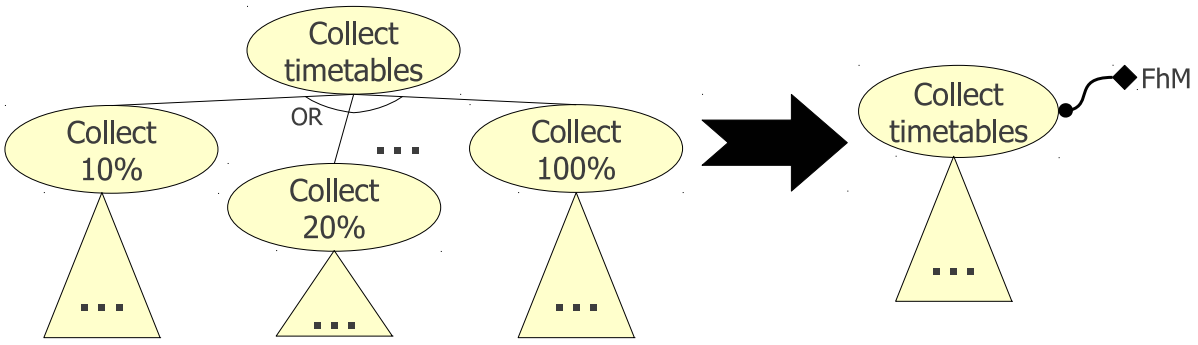


Figure 4.2: Using a control variable as an abstraction over families of subgraphs.

applied to goals, tasks and domain assumptions to represent abstractions over goal/domain model fragments. In particular, CVs are derived from families of related, but slightly different goal/task or domain assumption alternatives, as in Figure 4.2, where the goals *Collect timetables from 10% of participants*, *Collect timetables from 20% of participants*, etc. are shown as alternative ways to achieve the parent *Collect timetables* goal.

Here, we identify variations that differ in some value (usually, but not necessarily numeric) and abstract that value as a parameter to be attached to the appropriate goal model element as a CV (e.g., the FhM, *From how Many* variable in Figure 4.2). Figure 4.1 shows more examples of CVs, such as: RF (*Required Fields* when characterizing a meeting), RfM (number of *Rooms for Meeting* available — note that this CV applies to a domain assumption), etc.

As can be seen in both figures, a control variable is represented by a black lozenge (a diamond), attached to the goal model element to which it relates. For instance, MCA is attached to task *Let system schedule* (Figure 4.1) because it represents the *Maximum number of scheduling Conflicts Allowed* when the system looks for a suitable date/time to schedule the meeting. The analogy behind the choice of the diamond as graphical representation comes from the Java™ programming language, in which angle brackets are used to delimit the parameter for a generic type. When put together — <> — they are called the "diamond operator".[1]

The benefits of having CVs include the ability to represent large number of model variations in a compact way as well as the ability to concisely analyze how changes in CV values affect the system's success rate and/or quality of service when, e.g., scheduling meetings. As any parameter in software design, a CV needs to be taken into consideration (i.e., propagated) when refining the goal model element that it applies to and later when designing and implementing the system. In our approach, we are interested in analyzing

[1]See `http://docs.oracle.com/javase/tutorial/java/generics/gentypes.html`.

the effect of values of CVs on system output and thus omit the details of CV refinement and implementation.

Finally, *indicators* are essential to control systems as these are monitored system output values that feedback loops need to compare to the output targets in order to calculate the control error and to determine how the system's control input needs to be adjusted. Indicators are similar to *gauge variables*, proposed by van Lamsweerde [2009], and need to be measurable quantities.

In goal models, quality constraints as well as the success rates for hard goals and tasks can be used as indicators. Since the number of potential indicators is large, we need to select as indicators the important values that the adaptive system should strive to achieve. Therefore, we propose to use *Awareness Requirements*, introduced back in Section 3.1 (p. 64), as the set of system indicators that should be monitored.

Given the above definitions for system parameters and indicators and taking the *Find local room* goal of the Meeting Scheduler as an example, we would like to model information such as: "upon increasing the value of RfM, the success rate of *Find a suitable room* (referred to by *AwReq AR4*) also increases" and "at *VP2*, when choosing *Call hotels and convention centers* over *Call partner institutions*, your cost will increase (i.e., *AwReq AR2* might fail)". This kind of information is very important for a feedback controller in its task of deciding how to adapt the system to fulfill its requirements. The following sub-sections explain how to represent this information in the requirements specification.

4.1.2 Relations concerning numeric parameters

Numeric parameters can assume any integer or real value at runtime, but can have its range constrained by the problem domain. Three of the five control variables of the Meeting Scheduler, shown in Figure 4.1, are numeric, namely:

- FhM (attached to goal *Collect timetables*) indicates *From how Many* people the scheduler should collect timetables before this goal is considered satisfied. FhM accepts integer values in the range $[0, 100]$, representing a percentage value;
- RfM (attached to domain assumption *Local rooms available*) indicates the number of *Rooms for Meetings* available in the organization. Obviously, RfM is also integer and accepts only positive values;
- MCA (attached to task *Let system schedule*) indicates the *Maximum Conflicts Allowed* when the system is deciding the best date/time for the meeting. A conflict consists of a participant already having another appointment at the same time as the proposed date/time for the meeting. Therefore, like RfM, MCA should be a positive integer.

Changing the value of a numeric parameter affects many aspects of system performance, which, as explained in the previous sub-section, are measured through *indicators*. Taking the parameter RfM as an example, and assuming the success rate (the truth value) of *Local rooms available* is affected by changes in RfM, we could define this indicator as a function of the parameter (which is clearly a simplification):

$$success\ rate\ of\ Local\ rooms\ available = f(RfM) \tag{4.1}$$

We could then say how changes in RfM affect the success rate of the domain assumption by declaring if the derivative of f is positive or negative. Using Leibniz's notation:

$$\frac{\Delta\langle success\ rate\ of\ Local\ rooms\ available\rangle}{\Delta RfM} > 0 \tag{4.2}$$

Equation (4.2) tells us that if we increase the value of RfM, the success rate of *Local rooms available* also increases. Of course, the analogous decrease-decrease relation is also inferred. The $\Delta y/\Delta x$ notation is used instead of dy/dx because RfM, as previously mentioned, assumes only discrete values.

In practice, however, we use a simplified linearized notation (which always uses the Δ symbol) to improve writability, referring not to the success rates of requirements but to particular *AwReqs* that talk about their success/failure. In the case of *Find local room*, that *AwReq* is *AR8* and, therefore, Equation (4.3) shows how this relation would be actually specified in our approach:

$$\Delta\,(AR8/RfM) > 0 \tag{4.3}$$

Suppose now there is a limit to which this relation holds: after a given number, adding more rooms will not help with the success rate of *Local rooms available* (there are so many rooms that the organization could never occupy them all at the same time). For this case, we use the concept of *landmark values* (cf. Section 2.1.7, p. 45) and specify an interval in which the relation between the parameter and the indicator holds. Since we are dealing with qualitative information, we might not know exactly how many rooms are enough, so we define a landmark value called $enoughRooms$:

$$\Delta\,(AR8/RfM)\,[0, enoughRooms] > 0 \tag{4.4}$$

Although specifying this interval intuitively tells us that adding extra rooms after there are already enough of them available does not change the success rate of the goal, one could formalize this information, making it explicit:

$$\Delta\,(AR8/RfM)\,[enoughRooms, \infty] = 0 \tag{4.5}$$

This gives us the general form for differential relations in our proposal, shown in Equation (4.6), where the interval $[a, b]$ is optional, with default value $[-\infty, \infty]$, $\langle op \rangle$ should be substituted by a comparison operator ($>$, $\geq$, $<$, $\leq$, $=$ or $\neq$) and C is any constant, not just zero as in previous examples:

$$\Delta\left(indicator/parameter\right)[a, b] \ \langle op \rangle \ C \tag{4.6}$$

Non-zero values for C are useful for expressing different rates of change. When facing a decision on how to improve an indicator I, given the information $\Delta\left(I/P_1\right) > 0$ and $\Delta\left(I/P_2\right) > 0$ the controller would arbitrarily choose to either increase P_1 or P_2; on the other hand, $\Delta\left(I/P_1\right) > 2$ and $\Delta\left(I/P_2\right) > 7$ could help it choose P_2 in case I needs to be increased by a larger factor. Later, in Section 4.1.4, we also provide a different way of comparing the effect of changes of parameters that relate to the same indicator.

If we replace the constant C by a function $g(P)$, where P is the related parameter, we will be able to represent nonlinear relations between indicators and parameters, for instance, $\Delta\left(I/P\right) = 2 \times P$ (indicator I increases by the square of the increase of parameter P). However, linear approximations greatly simplify the kind of modeling we are proposing and are considered enough for our objectives. Moreover, it is very hard to obtain such precise qualitative values before the system is in operation.

4.1.3 Relations concerning enumerated parameters

In addition to numeric parameters, parameters that constrain their possible values to specific enumerated sets are also possible. Variation points are clear examples of this type of parameter, as their possible values are constrained to the set of paths in the OR-refinement. Control variables, however, can also be of enumerated type (in effect, as discussed earlier in Section 4.1.1, control variables are abstractions over families of goal models in an OR-refinement).

Figure 4.1 shows seven enumerated parameters elicited for the meeting scheduler, two enumerated control variables and three variation points:

- *RF* (attached to task *Characterize meeting*) indicates the *Required Fields* when the meeting is being characterized. It can assume the values: *participants list only*, *short description required* or *full description required*;
- *VPA* (attached to task *Collect from system calendar*) indicates if the meeting organizer is allowed to *View Private Appointments* of other employees when scheduling a meeting. It can be either *yes* or *no*;
- At *Collect timetables*, *VP1* can assume values *Call participants*, *Email participants* or *Collect automatically*;

- At *Find a suitable room*, *VP2* can assume values *Find local rooms*, *Call partner institution* or *Call hotels and convention centers*;
- At *Choose schedule*, *VP3* can assume values *Schedule manually* or *Let system schedule*;
- At *Find a local room*, *VP4* can assume values *Get room suggestions* or *List available rooms*;
- Finally, at *Book room*, *VP5* can assume value *Use available room* or *Cancel less important meeting*.

Unlike numeric parameters, the meaning of "increase" and "decrease" is not defined for enumerated types. However, we use a similar syntax to specify how changing parameter P from one value (α) to another (β) affects a system indicator I:

$$\Delta\left(I/P\right)\left\{\alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2, \ldots, \alpha_n \rightarrow \beta_n\right\}\ \langle op \rangle\ C \tag{4.7}$$

By performing pair-wise comparisons of enumerated values, stakeholders can specify how changes in an enumerated parameter affect the system. For example, the relations below show how changes in *VP2* affect, respectively, the indicators *AR2* (related to scheduling cost) and *AR10* (related to scheduling speed). When performing the changes represented between curly brackets, the success rate of *AR2* decreases (i.e., cost increases) whereas the success of *AR10* increases (i.e., speed increases).

$$\Delta\left(AR2/VP2\right)\left\{local \rightarrow partner, local \rightarrow hotel, partner \rightarrow hotel\right\} < 0 \tag{4.8}$$

$$\Delta\left(AR10/VP2\right)\left\{partner \rightarrow local, hotel \rightarrow local, partner \rightarrow hotel\right\} > 0 \tag{4.9}$$

Often, however, an order among enumerated values with respect to different indicators can be established. For instance, analyzing the pair-wise comparisons shown in relations (4.8) and (4.9), we conclude that for *AR2*, $local \prec partner \prec hotel$, whereas for *AR10* $partner \prec hotel \prec local$. Depending on the size of the set of values for an enumerated parameter, listing all pair-wise comparisons using the syntax specified in Equation (4.7) may be tedious and verbose. If it is possible to specify a total order for the set, doing so and using the general syntax presented for numeric parameters in Equation (4.6) can simplify elicitation and modeling.

This ordering specification $\alpha_1 \prec \alpha_2 \prec \ldots \prec \alpha_n$ can be either associated with a specific relation or defined as the default order for a specific enumerated parameter, to be applied to all of its relations unless specifically stated otherwise. For variation points, the default order is considered to be their position in the goal model, ascending from left to right. For instance, the default order of *VP2* is $local \prec partner \prec hotel$.

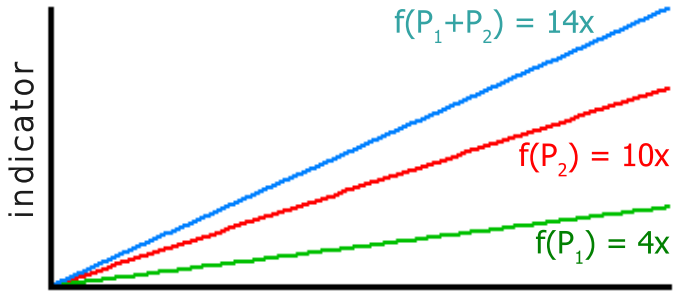


Figure 4.3: Combining the effects of different parameters on the same indicator.

4.1.4 Refinements and extrapolations

Differential relations always involve one indicator, but may involve more than one parameter. For example, "increasing" *VP1* and *VP3* (considering their default ordering) contributes positively to indicator *AR10* (which refers to *Schedules produced in less than a day*) both separately — $\Delta(AR10/VP1) > 0$ and $\Delta(AR10/VP3) > 0$ — and in combination — $\Delta(AR10/\{VP1, VP3\}) > 0$.

When we are not given any equation that differentially relates two parameters P_1 and P_2 to a single indicator I, we may still be able to extrapolate such a relation on the basis of simple linearity assumptions. For example, if we know that $\Delta(I/P_1) > 0$ and $\Delta(I/P_2) > 0$, it would be reasonable to extrapolate the relation $\Delta(I/\{P_1, P_2\}) > 0$. More generally, our extrapolation rule assumes that homogeneous impact is additive, as in Figure 4.3. Note that in cases where P_1 and P_2 have opposite effects on I, nothing can be extrapolated because of the qualitative nature of our relations.

Generalizing, given a set of parameters $\{P_1, P_2, \ldots, P_n\}$, if $\forall i \in \{1, \ldots, n\}$, $\Delta(I/P_i)\,[a_i, b_i]\,\langle op\rangle C$, our extrapolation rule has as follows:

$$\Delta(I/\{P_1, P_2, \ldots, P_n\}) \bigcap_{i=0}^{n} [a_i, b_i] \;\; \langle op\rangle C \tag{4.10}$$

If it is known that two parameters cannot be assumed to have such a combined effect, this should be explicitly stated, e.g., $\Delta(I/\{P_1, P_2\}) < 0$. For instance, in the Meeting Schedule we have the case in which both RfM and *VP2* contribute positively to indicator *AR4* (which refers to goal *Find a suitable room*) when increased, but both of them should not be increased at the same time. In other words, it does not make sense to increase the number of local rooms and then choose to use a hotel or partner institution, as you will only get the effect of the latter. The two relations and their combination are represented as follows (absolute values are used in comparisons between Δ-relations in order to properly

compare positive and negative effects, when applicable):

$$\Delta\left(AR4/RfM\right) > 0 \tag{4.11}$$

$$\Delta\left(AR4/VP2\right) > 0 \tag{4.12}$$

$$\left|\Delta\left(AR4/\left\{RfM, VP2\right\}\right)\right| = \left|\Delta\left(AR4/VP2\right)\right| \tag{4.13}$$

Although not explored in this thesis, other extrapolations are also possible. From differential calculus we could extrapolate on the concept of the second derivative. If $y = f(x)$, we can say that y grows linearly if $f'(x) > 0$ and $f''(x) = 0$ (it "has constant speed"). However, if we have $f''(x) > 0$, then y's rate of growth also increases with the value of x (it "accelerates"). Qualitative information on second derivatives could be modeled in our language using the following notation: $\Delta^2\left(I/P\right)\left[a,b\right]\left\langle op\right\rangle C$. Thus, if we say that $\Delta^2\left(I/P_1\right) > 0$ and $\Delta^2\left(I/P_2\right) = 0$, the controller may conclude that P_1 is probably a better choice than P_2 for large values. Other concepts, such as *inflection* and *saddle points*, *maxima* and *minima*, etc. could also be borrowed, although we believe that knowing information on such points in a $I = f(P)$ relation without knowing the exact function $f(P)$ is very unlikely.

4.1.5 Differential relations for the Meeting Scheduler

The indicators, parameters and some of their relations for the Meeting Scheduler example have been presented earlier, throughout this chapter. However, for completeness, Table 4.1 presents equations (4.14)–(4.49), which represent all of the differential relations elicited for the example, except for those of meta-*AwReqs* *AR13* and *AR14*, which are exactly the same as the *AwReqs* to which they refer, respectively, *AR7* and *AR6*.

Notice that there is no parameter that affects *AwReqs* *AR11* and *AR12* and, thus, reconfiguration is not an option for a failure of these indicators. Not coincidentally, back in Section 3.2 (p. 78) we have associated them to *adaptation strategies* that use evolution instead of reconfiguration. In the next section, we associate the *reconfiguration* strategy to other Meeting Scheduler *AwReqs*, to which it is possible to do so.

4.2 Qualitative adaptation specification

According to Wang and Mylopoulos [2009], a system *configuration* is "a set of tasks from a goal model which, when executed successfully in some order, lead to the satisfaction of the root goal". We add to this definition the values assigned to each *control variable* elicited during system identification (cf. Section 4.1). *Reconfiguration*, then, is the act of replacing the current configuration of the system with a new one in order to adapt. Some existing reconfiguration approaches were summarized back in Section 2.2.3 (p. 50).

Table 4.1: Differential relations elicited for the Meeting Scheduler example.

$$order\,(RF) : listonly \prec short \prec full \tag{4.14}$$

$$order\,(VP2, AR10) : partner \prec hotel \prec local \tag{4.15}$$

$\Delta\,(AR1/RF) < 0$ (4.16)	$\Delta\,(AR6/VP1) < 0$ (4.28)
$\Delta\,(AR2/RfM) < 0$ (4.17)	$\Delta\,(AR6/VP3) < 0$ (4.29)
$\Delta\,(AR2/VP2) < 0$ (4.18)	$\Delta\,(AR7/VPA)\,\{false \rightarrow true\} < 0$ (4.30)
$\Delta\,(AR3/FhM) < 0$ (4.19)	$\Delta\,(AR8/RfM)\,[0, enough] > 0$ (4.31)
$\Delta\,(AR4/RfM) > 0$ (4.20)	$\Delta\,(AR8/VP2) > 0$ (4.32)
$\Delta\,(AR4/VP2) > 0$ (4.21)	$\Delta\,(AR9/MCA) > 0$ (4.33)
$\Delta\,(AR5/MCA) > 0$ (4.22)	$\Delta\,(AR9/VP3) > 0$ (4.34)
$\Delta\,(AR5/VP3) < 0$ (4.23)	$\Delta\,(AR10/RF) < 0$ (4.35)
$\Delta\,(AR6/RF) > 0$ (4.24)	$\Delta\,(AR10/FhM) < 0$ (4.36)
$\Delta\,(AR6/FhM) > 0$ (4.25)	$\Delta\,(AR10/VP1) > 0$ (4.37)
$\Delta\,(AR6/VPA)\,\{false \rightarrow true\} > 0$ (4.26)	$\Delta\,(AR10/VP2) > 0$ (4.38)
$\Delta\,(AR6/MCA) < 0$ (4.27)	$\Delta\,(AR10/VP3) > 0$ (4.39)

$$|\Delta\,(AR2/RfM)\,| < |\Delta\,(AR2/VP2)\,| \tag{4.40}$$

$$|\Delta\,(AR4/RfM)\,| = |\Delta\,(AR4/VP2)\,| \tag{4.41}$$

$$|\Delta\,(AR4/\{RfM, VP2\})\,| = |\Delta\,(AR4/VP2)\,| \tag{4.42}$$

$$|\Delta\,(AR6/VPA)\,| < |\Delta\,(AR6/RF)\,| < |\Delta\,(AR6/VP3)\,| < |\Delta\,(AR6/FhM)\,| < \ldots \tag{4.43}$$

$$\ldots < |\Delta\,(AR6/FhM)\,| < |\Delta\,(AR6/VP1)\,| < |\Delta\,(AR6/MCA)\,| \tag{4.44}$$

$$|\Delta\,(AR8/RfM)\,| = |\Delta\,(AR8/VP2)\,| \tag{4.45}$$

$$|\Delta\,(AR8/\{RfM, VP2\})\,| = |\Delta\,(AR8/VP2)\,| \tag{4.46}$$

$$|\Delta\,(AR9/MCA)\,| < |\Delta\,(AR9/VP3)\,| \tag{4.47}$$

$$|\Delta\,(AR10/RF)\,| < |\Delta\,(AR10/VP2)\,| < |\Delta\,(AR10/VP3)\,| < \ldots \tag{4.48}$$

$$\ldots < |\Delta\,(AR10/VP3)\,| < |\Delta\,(AR10/FhM)\,| < |\Delta\,(AR10/VP1)\,| \tag{4.49}$$

Awareness Requirements (cf. Section 3.1, p. 64), used as indicators, coupled with parameters and differential relations, provide us enough information to reason over the goal model to find out, when there is an *AwReq* failure, which (if any) parameters can be changed (and to which direction, i.e., increase or decrease) in order to adapt, effectively reconfiguring the system in response to failures.

Given what we have so far, then, a possible reconfiguration algorithm could be composed of three steps: given an *AwReq* failure, (1) find all system parameters that affect the *AwReq* positively; (2) calculate the one(s) with the least negative impact on other indicators; and (3) return a new system configuration changing the value of this/these parameter(s).

However, with the above algorithm, there are still a few information missing regarding the requirements for this adaptation step. For instance, how many parameters should be changed and by how much? When calculating negative impact to other indicators, should priorities among them be considered? What if the *AwReq* fails again, should the previous attempts to reconfiguration be taken into account when deciding a new one?

As discussed back in Section 2.1.7 (p. 45), in Qualitative Reasoning different representation languages provide different levels of precision. In the design of adaptive systems, different information on indicators, parameters and their relations may also come in different precision levels. For example, Table 4.1 shows that, for the Meeting Scheduler, we are able to establish an order of effectiveness for some indicators (e.g., Equation (4.40) about *AR2*), whereas for others either we are not able to do so (e.g., *AR5*) or we explicitly indicate that effects are equal (e.g., Equation (4.42) about *AR4*). The result is that an adaptation algorithm can be more precise when dealing with indicators that do establish this order, but for others the choice of parameter to change may have to be random or arbitrary.

Given that the availability of more precise information can vary from one system to another or even change in time for the same system, we propose to complement the specification of qualitative relations among indicators and parameters with requirements for the reconfiguration of the system, consisting of assigning possibly different *adaptation algorithms* to different *AwReq* failures. These algorithms are composed of procedures for eight different activities that form a framework for run-time qualitative adaptation. Different procedures require different information from the requirements specification, accommodating varying levels of precision. Furthermore, the framework is extensible, allowing for the creation of new procedures for particular cases.

In this section, we describe this framework for qualitative adaptation, present some procedures and algorithms that have already been defined for it and, finally, close the loop on the reconfiguration capabilities of the adaptive Meeting Scheduler through the

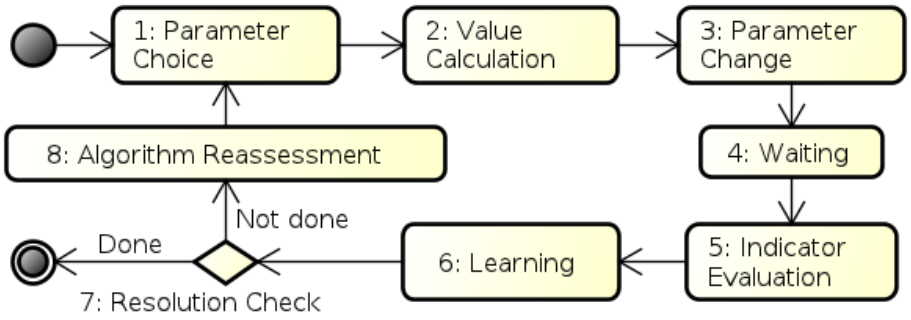


Figure 4.4: The adaptation process followed by the *Qualia* framework.

selection of adaptation algorithms for each *AwReq* failure.

4.2.1 A framework for qualitative adaptation

As seen in Section 4.1, system identification adds to a requirements model qualitative information on how changes in system parameters affect indicators that are deemed important by the stakeholders. In this thesis, we propose a framework to operationalize adaptation at runtime based on this qualitative information. We call this framework *Qualia* (**Quali**tative **a**daptation) and detail it in Chapter 6. In this section, we provide an overview of *Qualia* in order to understand the information that needs to be added to the requirements models in order to specify how the system should reconfigure in response to failures.

When made aware of a failure in an indicator, *Qualia* adapts the system by conducting eight activities, as shown in Figure 4.4: one or more parameters modeled during system identification are chosen (1), then, based on the relation of these parameters with the failed indicator, *Qualia* decides by how much they should be changed (2). The parameters are then incremented (consider decrements as negative increments for simplification) by this value (3) and the framework waits for the change to produce any effect on the indicator (4), evaluating it again after the waiting time (5). In each cycle, *Qualia* can learn from the outcome of this change, possibly evolving the adaptation mechanism and updating the model (6). Finally, it decides whether the current indicator evaluation is satisfactory (7) and either concludes the process or reassesses the way it was conducted in the previous cycles (8) and starts over.

To accommodate the different levels of precision, we propose an extensible framework by defining an interface for each activity in the process of Figure 4.4, providing default implementations that assume only the minimum amount of information is available and allowing designers to create and plug-in new *procedures* into *Qualia*, possibly requiring more information about the system in order to be applicable.

We use the term *adaptation algorithm* to refer to the set of procedures chosen to support the adaptation process. An important remark here is that we do not make any claim on which adaptation algorithm is better suited for any particular context. Instead, the different implementations presented in this section serve to illustrate how this framework can be extended as needed. The choice of algorithm to use is the responsibility of the analysts.

Therefore, the information that should be encoded in the requirements specification is which adaptation strategy to use to adapt to every specific (*AwReq*) failure. When adapting the system, *Qualia* selects for each activity of its process the procedure that has been specified for the given situation according to stakeholder preferences. In the absence of such specification, however, the framework uses default procedures which require minimum information from the requirements models, namely:

- **Indicators:** *Qualia* has to be notified of indicator failure, hence the model should specify what are the relevant indicators in a way such that another component of the feedback loop is able to monitor them. As already stated earlier, our approach uses *AwReqs* as indicators;

- **Parameters:** to adapt to an indicator failure, there should be at least one related parameter. Section 4.1 has described how this information is specified through differential relations;

- **Unit of increment:** each numeric parameter must specify its unit of increment, because *Qualia* will not be able to guess it.

The *unit of increment* is also important for the comparison among indicator/parameter relations. For instance, the comparison $|\Delta(AR6/FhM)| < |\Delta(AR6/MCA)|$ presented in the context of the Meeting Scheduler earlier in Table 4.1, should be complemented with $U_{FhM} = 10\%$ and $U_{MCA} = 1$, meaning that changing MCA (*Maximum Conflicts Allowed* when scheduling) *by 1 conflict* improves *AR6* more than changing FhM (*From how Many* participants timetables should be collected) *by 10 percent*. Moreover, enumerated parameters must be ordered (cf. Section 4.1.3) and their unit of increment defaults to choosing the next value in the order.

Given this information, the *Default Adaptation Algorithm* starts by using a *Random Parameter Choice* procedure that picks one parameter randomly from the set of parameters related to the failed indicator, considering those which can still be incremented by at least one unit (i.e., are within their boundaries). Taking the Meeting Scheduler as an example, imagine that in the past month, less than 75% of the meetings had *Good participation*, breaking *AwReq AR6*. Available parameters to improve this indicator are VPA,

RF, $VP3$, FhM, $VP1$ and MCA (assuming all within boundaries). For this running example, consider the random procedure chose FhM.

Next, *Qualia* decides the increment value for the chosen parameter. In the default algorithm, a *Simple Value Calculation* procedure multiplies the value of the parameter's unit of increment U by the indicator's increment coefficient K, returning the increment value $V = K \times U$. The increment coefficient is an optional parameter (with default value $K = 1$) that can be associated to each indicator in the specification to determine how critical it is to adapt to their failures. Higher values of K will produce more significant changes, but the requirements engineer should be aware of the risks of overshooting when defining them. Note also that parameters should never exceed their boundaries. In the Meeting Scheduler example, say $K_{AR6} = 2$ and we know that $U_{FhM} = 10\%$ and $\Delta\,(AR6/FhM) > 0$, so *Qualia* should increase FhM by $V = 2 \times 10\% = 20\%$.

Afterwards, the *Simple Parameter Change* procedure changes the chosen parameter by the calculated value, at the *class* level, meaning that the changes will affect the system "from this point on". On the other hand, if the analyst specifies that the change should be done at the *instance* level, these would only affect the current execution of the system. This terminology is inherited from the specification of requirements as UML classes, as detailed back in Section 3.1.2 (p. 70). In the example, FhM is increased by 20% for all meeting schedules produced after the *AR6* failure, until further notice.

Following parameter change, the framework waits for some time and evaluates the indicator. The *Simple Waiting* procedure is to wait until the next time the indicator is evaluated. For instance, *AR6* is evaluated at every month. After the waiting time, *Qualia* executes the *Boolean Indicator Evaluation* procedure, simply verifying if this time the indicator succeeded, e.g., if after FhM was increased by 20%, in the next month at least 75% of the meetings had *Good participation*.

In the default algorithm, the learning step is skipped (*No Learning*) and *Qualia* moves on to the final two steps: deciding whether to stop or iterate and, in the latter case, reassessing the strategies. The *Simple Resolution Check* procedure stops the process if the outcome of the indicator evaluation was positive. Otherwise, it iterates, using the *No Algorithm Reassessment* procedure which, as the name indicates, always keeps the same algorithm for the following cycle of the process. Finalizing the Meeting Scheduler example, if the 20% reduction was effective the process will stop; otherwise it will repeat the same procedures, as above.

In the requirements specification, *adaptation algorithms* should be represented by a set that specifies which non-default procedures should be used. Therefore, the *Default Adaptation Algorithm* can be represented by the empty set $\varnothing$, meaning all the default procedures described above will be used. The following sub-section presents alternative

procedures that can compose more elaborate adaptation strategies, showing how *Qualia* can be extended as needed.

4.2.2 Accommodating precision

As stated before, *Qualia* supports different levels of precision by allowing for new procedures to be implemented and plugged in the framework for each of the eight activities in its adaptation process. Here, we illustrate a few alternative procedures that can be used to compose different adaptation algorithms, then two specific adaptation algorithms that combine procedures in order to: (a) converge to optimal parameter values; and (b) execute the PID algorithms described back in Section 2.1.5 (p. 39). For the alternative procedures, we focus here on the *Parameter Choice* activity and describe new procedures that execute it differently from the default one, especially in the presence of more precise information in the specification.

The *Random Parameter Choice* procedure, which is part of the default algorithm, selects a single parameter from the specification in a random fashion. With the same amount of information from the model, we can implement a *Shuffle Parameter Choice* procedure, which randomly puts the system parameters in order during the first cycle and picks the next one using this predefined sequence when switching parameters is required (more on the *repeat policy* later). However, if differential relations regarding the indicator in question have been refined to provide comparison of their effect (as explained in Section 4.1.4), this procedure can be refined to an *Ordered Effect Parameter Choice*, which orders the parameters according to their effect on the indicator, either in ascending or descending order (depending if stakeholders would like to start with the parameters that have the greatest or the smallest effect on the indicator). If comparison of effect is not provided for all parameters, the analyst should specify if the remaining parameters should be excluded from the sequence or shuffled at the end of it.

All of the procedures presented above can be further customized by some attributes, one of which is the *number of parameters* to choose. This attribute, as can be seen from the past descriptions, defaults to *one*, but can be set to any positive integer, or even *all* parameters, mimicking the behavior of a *multiple input, single output* control system. Another attribute is the *repeat policy*, whose default value — *repeat when incrementable* — tells the procedure to repeat the parameter chosen in the previous cycle until it is has reached the boundary set by its relation with the indicator. Other values can be *repeat M times*, where M is configurable and *repeat while oscillating*, which will be explained in the following subsection.

More precise information in the specification can also benefit the default *Waiting* procedure presented earlier. Relations' *maturation times* (optional) indicate how long it

takes for the changes in the related parameter to take effect in the related indicator. For instance, consider a different scenario of the Meeting Scheduler in which *AR8*, referring to domain assumption *Local rooms available*, fails and parameter RfM (*Rooms for Meetings*) is selected to reconfigure the system. Preparing a new room so it becomes available for meetings might take a while and, therefore, the adaptation framework should wait for this specified time before continuing.

Back to the *Parameter Choice* activity, such new attributes provide yet another way of ordering parameters, so a new procedure could be proposed for choosing first the parameters with lowest *maturation time*, i.e., the fastest acting parameters. As we can see, our proposed framework can be extended as needed by requirements engineers, depending on stakeholder requirements and the availability of information about the system and its environment.

Overshoot avoiding algorithms

One of the desired characteristics of control systems is to avoid overshooting its control inputs. For instance, if the success rate of *Good participation* for a given month is 70% (breaking *AR6*), we decide to increase FhM from 60% to 80% and, in the following month, the success rate becomes 85%, we have overshot the improvement on the indicator by 10%. Granted, this overshoot could be corrected whenever some other indicator (e.g., *AR10*, which controls if scheduling is done quickly) fails and FhM is chosen to be decremented. Still, a good adaptation algorithm tries to avoid overshooting in the first place and, in what follows, we present one such algorithm.

The *Oscillation Algorithm* works as depicted in Figure 4.5: back to the *AR6* / FhM scenario, imagine that given the current circumstances, the optimal[2] value for FhM is 70%. The controller obviously does not know it, so when *AR6* fails, it increases FhM from the current value of 55% to 75%, which actually solves the problem.

However, instead of stopping here, the algorithm **assumes to have overshot the change**, and thus starts changing the same parameter in the opposite direction, using **half of the previous increment value**. When FhM is set back to 65%, *AR6* fails again, which makes the controller switch increment direction and halve the increment value one more time. This process goes on until one of the following conditions:

- The parameter is incremented to a value that it has already assumed before, which means that we should be very close to the optimal value. For example, if we continue

[2]Here, we consider "optimal" the smallest change that fixes the problem, because we assume every adaptation brings negative side effects to other indicators. If this is not the case, one could just set the parameter to its maximum (minimum) value from the start and no adaptation is necessary!

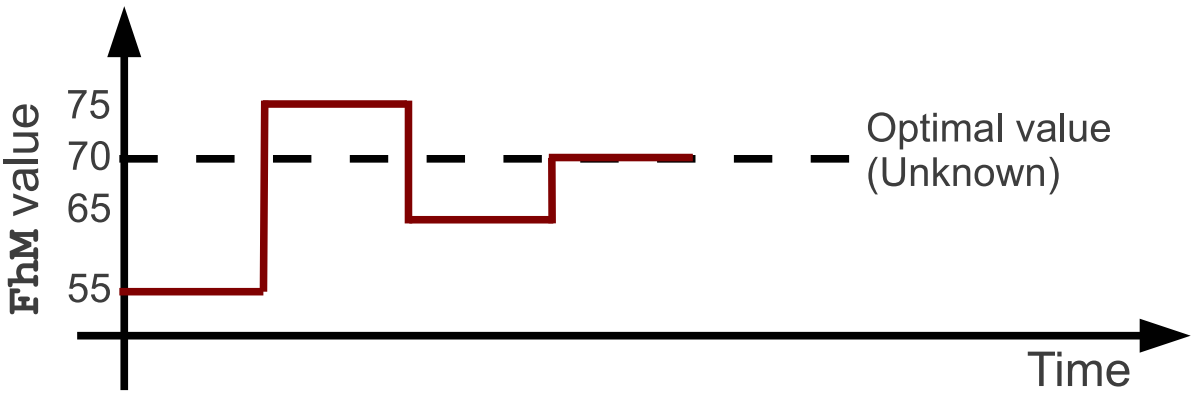


Figure 4.5: A scenario of use of the *Oscillation Algorithm* in the Meeting Scheduler.

the oscillations shown in Figure 4.5, FhM will assume values 67%, 69%, 71%, 70% and then stop;

- The algorithm has already performed the *maximum number of oscillations*, which is an optional attribute that can be assigned to a specific *AwReq* or to the entire goal model. Here, we consider each inversion of direction to be an oscillation (three, in the figure);

- The increment value is halved to an amount that is lower than the *minimum change value* of the parameter at hand (optional). For instance, Figure 4.5 represents the case in which this value is 5%. Note that, for integer variables such as FhM, 1 is the lowest possible value.

In order to tune this algorithm, the framework also allows for the specification of parameters' *halving factors* different from the default value of 0.5. When oscillating, the increment value will be multiplied by the specified factor. Furthermore, the selection of *Parameter Choice* procedure, as we saw in the previous subsection, defines how many parameters will be tuned and in which order. One *Oscillation Algorithm* could then be represented by the set {*Ordered Effect Parameter Choice*, *Oscillation Value Calculation*, *Oscillation Resolution Check*}, tuning parameters in ascending (or descending) order of effect. The important detail here is to set the *Parameter Choice* procedure's *repeat policy* to *repeat while oscillating*, which will tell the framework to choose the same parameter until one of the above stop conditions. When working with multiple parameters, one could aim for the least possible overshoot, in which **all parameters but the last** (in descending order, the one with least effect) are tuned to the value that is closest to the optimal, but still leaves the indicator in the failed state.

A PID-based adaptation algorithm

In the above, we have proposed algorithms that were inspired by the PID controller. The question that arises then is the following: considering single outputs, and sometimes also single inputs, would it be possible to use the actual PID algorithm, as described in Section 2.1.5 (p. 39), in our models? Since this algorithm requires a numeric value for the control error and *AwReqs* are somewhat of a Boolean nature ($success = true|false$), this question is then replaced by another one: can we extract a numeric control error from *AwReqs*?

As explained back in Section 3.1 (p. 64), *AwReqs* can be divided in three categories: *Delta AwReqs* impose constraints over properties of the domain (e.g., "*Manage meeting* should execute within an hour of the meeting chosen time"), *Aggregate AwReqs* determine requirements' success rates ("75% of the meetings should cost less than €100"), and *Trend AwReqs* impose constraints over aggregated success rates over time ("success rate of *Collect timetables* should not decrease two weeks in a row"). *Qualia* will extract numeric control errors from these types of *AwReqs* as follows:

- *Delta AwReqs*: if the property is numeric, calculate the difference between desired and monitored values. In the above example, it is the difference (in seconds) of the time *Manage meeting* was executed and the time it should have been executed (an hour after the meeting started);
- *Aggregate AwReqs*: calculate the difference between the desired and actual success rates. Note that *AwReqs* of the form "R should never fail" can be translated into "R should have 100% success rate";
- *Trend AwReqs*: calculate the difference between the last two measured success rates. In the above example, if the rate decreases in 7% in the first week and then again by 4% in the second, the control error is 4%.

If the *AwReq* in question follows one of these patterns, the *PID Algorithm* (i.e., {*PID Value Calculation*, *PID Indicator Evaluation*, *PID Resolution Check*}) can be used, which implements the algorithm described back in Section 2.1.5 (p. 39).

4.2.3 Specifying adaptation algorithms

Given the above algorithms, to close the loop on the reconfiguration capabilities of an adaptive system-to-be after *AwReqs*, parameters and differential relations have been elicited, requirements engineers should specify the adaptation algorithm to use for each *AwReq*, plus provide any information used by the chosen algorithm. Table 4.2 shows the

Table 4.2: New elements, w.r.t. the core requirements ontology [Jureta et al., 2008].

Element	**Used by**
Awareness Requirements (*AwReqs*) as indicators	Monitoring component
Control variables and variation points (parameters)	Parameter Choice
Differential relations between parameters and indicators	Parameter Choice, Parameter Change
Differential relations' refinements (comparison, cummulative effect)	Parameter Choice
AwReqs' increment coefficients	Value Calculation
Parameters' units of increment	Value Calculation
Relations' maturation times	Waiting
Global or *AwReqs*' maximum number of oscillations	Resolution Check
Parameters' minimum change values	Resolution Check
Parameters' halving factors	Parameter Change

complete list of new modeling elements that have been proposed so far in our research and the component/procedure that makes uses of that information.

We can now complement the requirements specification for the adaptive Meeting Scheduler with some of this information:

- **Increment coefficients**: $K_{AR2} = 2$, $K_{AR6} = 2$;
- **Units of increment**: $U_{FhM} = 10\%$, $U_{RfM} = 1\ room$, $U_{MCA} = 1\ conflict$;
- **Relations' maturation times**: $T_{\Delta(*/RfM)} = 2\ days$, $T_{\Delta(AR7/VPA)} = 15\ days$.

Finally, an adaptation algorithm can be specified for each possible *AwReq* failure of the Meeting Scheduler, as shown in Table 4.3. As explained back in Section 3.2.3 (p. 85), reconfiguration and evolution are unified into a single framework and specified as adaptation strategies. Therefore, Table 4.3 represents the final specification for the adaptation requirements of the Meeting Scheduler.

As mentioned before, $\varnothing$ represents the default algorithm and $\{P_1, \ldots, P_n\}$ represents an algorithm that uses the specified procedures $P_1, \ldots, P_n$ instead of their respective default ones, keeping the ones that were not replaced. Furthermore, algorithm properties such as the *order* to consider when choosing parameters and the number of parameters n to change are given between square brackets when applicable.

Table 4.3: Final specification for the Meeting Scheduler, including reconfiguration algorithms.

AwReq	***AwReq* pattern**	**Adaptation strategies**
AR1	`NeverFail(T_CharactMeet)`	1. *Retry(5000)* 2. *Reconfigure(∅)*
AR2	`SuccessRate(Q_CostLess100, 75%)`	1. *Reconfigure({Ordered Effect Parameter Choice})*
AR3	`not TrendDecrease(G_CollectTime, 7d, 2)`	1. *Replace(AR3, CLASS, AR3_3weeks)* 2. *Warning(IT_Staff)* 3. *Reconfigure(∅)*
AR4	`NeverFail(G_FindRoom)`	1. *Reconfigure(∅)*
AR5	`NeverFail(G_ChooseSched)`	1. *Reconfigure({Oscillation Value Calculation, Oscillation Resolution Check})*
AR6	`SuccessRate(Q_Min90pctPart, 90%, 1M)`	1. *Reconfigure({Ordered Effect Parameter Choice}[order = descending, n = 2])*
AR7	`NeverFail(D_ParticUseCal)`	1. *Reconfigure(∅)* 2. *Replace(CLASS, T_EnforceUseOfCalendar)*
AR8	`MaxFailure(D_LocalAvail, 1, 7d)`	1. *Delegate(Management)* 2. *Reconfigure(∅)*
AR9	`ComparableSuccess( T_SchedSystem, T_SchedManual, 10)`	1. *Reconfigure({Ordered Effect Parameter Choice}[order = descending])*
AR10	`@daily SuccessRate(Q_Sched1Day, 90%, 10d)`	1. *Reconfigure({Ordered Effect Parameter Choice}[order = ascending])*
AR11	—	1. *Warning(Secretary)*
AR12	`StateDelta(T_ConfirmOcc, Undecided, *, 5m)`	1. *Warning(Secretary)*
AR13	`SuccessRate(AR7, 80%)`	1. *Reconfigure(∅)*
AR14	`not TrendDecrease(AR6, 30d, 2)`	1. *Reconfigure({Ordered Effect Parameter Choice}[order = descending, n = 2])*

4.3 Chapter summary

In this chapter, we present the second contribution of this thesis, namely, mechanisms for qualitative adaptation through reconfiguration. There are two aspects of this contribution: a language for modeling qualitative information about the behavior of the system (§ 4.1) and the means for specifying, given a family of reconfiguration algorithms, which one should be used at each particular situation (§ 4.2).

The language that models system behavior is based on System Identification approaches from Control Theory and, thus, requires the identification of system parameters and indicators of requirements convergence (§ 4.1.1). Once these are identified, differential (Δ) relations can be used to represent how changes in parameters affect indicators, independently if such parameters are numeric (§ 4.1.2) or enumerated (§ 4.1.3). Moreover, refinements indicating if one parameter has greater effect than another with respect to an indicator and extrapolations on their combined use are also possible (§ 4.1.4).

Given the model of system behavior that has been just described, a family of reconfiguration algorithms is proposed by defining a process composed of eight steps, implemented by a default algorithm comprising default implementations for each of the steps of the process (§ 4.2.1). Being a qualitative approach, the default procedures requires very little information from the requirements model (basically the information provided by the aforementioned Δ-equations), but other procedures requiring higher levels of precision are provided and the framework can be extended as needed (§ 4.2.2).

As with the previous chapter, the concepts presented herein were also illustrated using the running example of the Meeting Scheduler, whose differential equations were presented in Section 4.1.5 and the list of adaptation algorithms to be used in Section 4.2.3. Table 4.3 (p. 110) complements the previously presented specification of monitoring and adaptation requirements for the Meeting Scheduler (cf. Table 3.7 in p. 87), representing its final specification.

Chapter 5

Designing adaptive systems

Simple things should be simple, complex things should be possible.

Alan Kay

The previous chapters presented new modeling elements that augment the state-of-the-art in Goal-Oriented Requirements Engineering in order to represent the requirements specification for adaptive systems based on a feedback loop architecture. The next research question, proposed in Chapter 1 as **RQ3**, is: *How can we help architectural designers and programmers implement this requirements-based feedback loop?*

The answer to this question is twofold: first, we need to aid requirements engineering in the process of going from "vanilla"[1] requirements to the specifications for adaptive systems illustrated in Chapter 3. Then, a run-time framework could implement the generic functionalities of the feedback loop based on the system's adaptation requirements, relieving the developers of this task and promoting reuse based on models that have a high-level of abstraction.

Such run-time framework will be the focus of Chapter 6, whereas in this chapter we detail our proposed systematic process for the design of adaptive systems. An overview of this process is presented next.

5.1 Overview

Figure 5.1 gives an overview of our proposed process for the design of adaptive systems, which we call *the Zanshin approach*. *Zanshin* (残心) is a Japanese term used in martial arts to represent a state of total awareness,[2] a reference to the first modeling element

[1] As before in this thesis, by "vanilla" we mean the requirements of the system-to-be that are not related to its desired adaptation capabilities.

[2] See `http://en.wikipedia.org/wiki/Zanshin`.

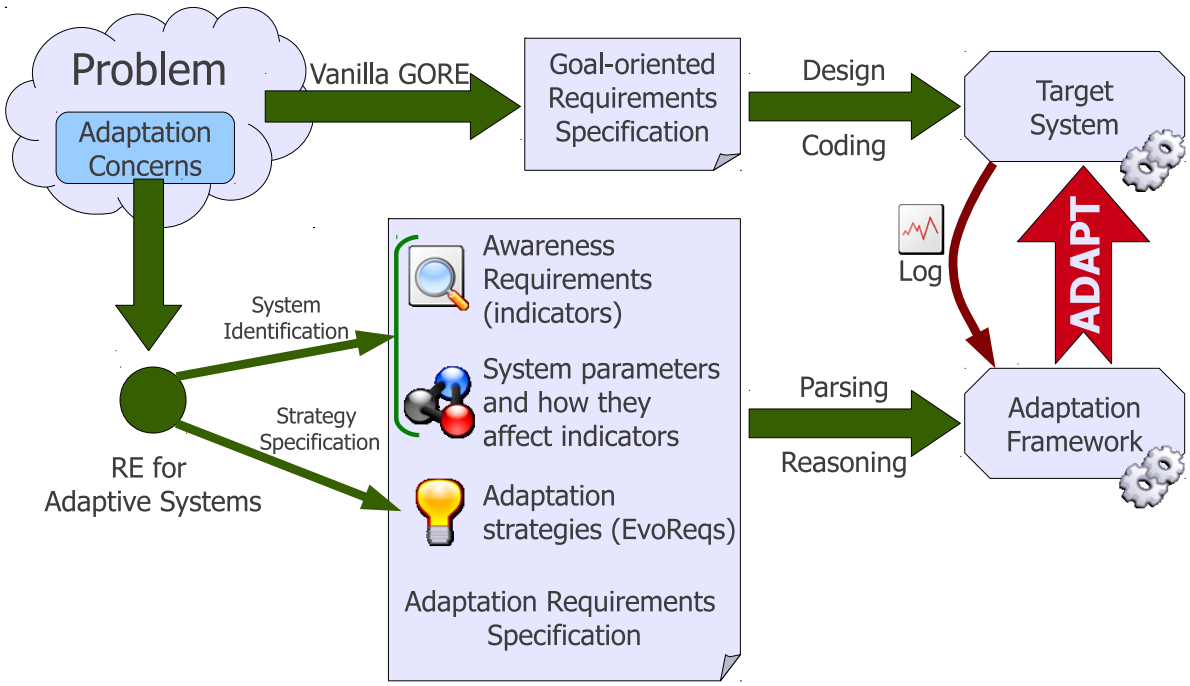


Figure 5.1: Overview of the *Zanshin* approach for the design of adaptive systems.

produced by the approach, *Awareness Requirements* (cf. Section 3.1, p. 64).

The approach divides the software development process in two tracks: *"Vanilla" Requirements Engineering, Design and Coding* is at the top, while *Requirements Engineering for Adaptive Systems* can be found at the bottom. This separation of concerns is merely conceptual, not processual, i.e., the adaptive part of RE is, of course, highly dependent on the (partial or final) results of the "vanilla" RE activity and they are not parallel, independent subprocesses, as the diagram might suggest.

The "vanilla" software development process is concerned with modeling the requirements, designing the architecture and coding the target system in the usual way, i.e., independently of aspects related to adaptation capabilities that the target system is supposed to have. For these activities, we do not prescribe any specific process or methodology. However, we do impose a few constraints on this process, some of which have already been mentioned before. These constraints are:

- Our framework is goal-oriented and expects as output of the "vanilla" RE phase a goal model of the system's requirements. This was discussed in Section 2.1.3 (p. 32);
- In order for the feedback loop implemented in the *Adaptation Framework* to perform run-time reasoning over the requirements model, the *Target System* should log (in a medium accessible by the framework) information about the execution of system

requirements. This will be discussed in more depth in Chapter 6, next;

- Finally, to adapt the *Target System*, the *Adaptation Framework* will send adaptation instructions that, as explained in Section 3.2 (p. 78), take the form of *Evolution Requirements* operations (e.g., `abort()`, `initiate()`, `rollback()`, etc.). The *Target System* has to be able to receive these instructions and operationalize them.

It is important to note that the process described in this chapter does not support legacy systems. Adding adaptation capabilities to existing systems for which models and/or source code are not available is considered future work in the context of this research.

Given that the "vanilla" software development process has provided the required artifacts described above, we propose a systematic process for the inclusion of adaptation features in the system-to-be. This process is composed of two main activities: *System Identification* and *Adaptation Strategy Specification*.

5.2 System Identification

As mentioned in the previous chapter, cf. Section 4.1 (p. 91), *System Identification* is the process of determining the equations that govern the dynamic behavior of a control system and is concerned with the identification of system parameters and the nature of their effect on the monitored output of the system. We propose to model the output to be monitored through Awareness Requirements (*AwReqs*), system parameters using variation points and control variables, and the effect of parameters on the output via differential relations.

In this section, we describe a systematic process for system identification of adaptive software systems. The input for this process is a requirements model G, such as the one depicted in Figure 2.6 (p. 37) for the Meeting Scheduler, containing elements such as goals, tasks, domain assumptions and quality constraints (cf. Section 2.1.3 in p. 32, also [Jureta et al., 2008]).

After the four steps of the process, namely indicator identification, parameter identification, relation identification and relation refinement, the output of this process is a parametrized specification of the system behavior $S = \{G, I, P, R(I, P)\}$, where:

- G is the initial goal model, provided as input;

- I is the set of indicators, identified in the model by *AwReqs*, which were characterized earlier in Section 3.1 (p. 64);

- P is the set of parameters, comprising both variation points and control variables, which were described back in Section 4.2 (p. 99);
- Finally, $R(I, P)$ is the set of relations between indicators and parameters, as also explained in Section 4.2.

Each step of the system identification process augments the specification S, adding or refining the information contained therein. In the following sub-sections, we provide more detail on each step of this process, which does not necessarily have to be conducted in a sequential fashion, but could also be applied iteratively, gradually enriching the model with each iteration.

5.2.1 Indicator identification

Input: the initial goal model G;

Output: partial specification of system behavior $S = \{G, I\}$.

As already mentioned, we propose to use Awareness Requirements (*AwReqs*) as indicators. Like other types of requirements, *AwReqs* must be systematically elicited. Since they refer to the success/failure of other requirements, their elicitation takes place after the basic requirements have been elicited and the goal model constructed (but could also be done iteratively).

There are several common sources of *AwReqs* and, in this section, we discuss some of these sources. We do not, however, propose any particular technique for the identification of *AwReqs* and requirements engineers should use existing requirement elicitation techniques to discover requirements that belong to this new class.

Critical requirements

One obvious source consists of the goals that are critical for the system-to-be to fulfill its purpose. If the aim is to create a robust and resilient system, then there have to be goals/tasks in the model that are to be achieved/executed at a consistently high level of success. Such a subset of critical goals can be identified in the process and *AwReqs* specifying the precise achievement rates that are required for these goals will be attached to them. Requirements that are controlled by regulations or Service Level Agreements (SLAs) are good candidates for *AwReq* targets.

This process can also be viewed as the operationalization of high-level non-functional requirements (NFRs) such as robustness, dependability, etc. For example, the task *Characterize meeting* is critical for the process of meeting scheduling since all subsequent

activities depend on it. Also, government regulations and rules may require that certain goals cannot fail or be achieved at high rates. Similarly, *AwReqs* are applied to domain assumptions that are critical for the system (e.g., *Participants use the system calendar*).

Non-functional requirements

The NFRs that are directly present in the goal model, in the form of softgoals, can also be sources of *AwReqs*. In the previous chapter, we presented a quality constraint *Meetings cost less than €100* that metricizes a high-level softgoal *Low cost*. Then, *AwReq AR2* is attached to it requiring the success rate of 75%. This way the system is able to quantitatively evaluate at runtime whether the quality requirements are met over large numbers of process instances and make appropriate adjustments if they are not.

In early requirements (cf. Section 2.1, p. 25), qualitative softgoal contribution labels in goal models capture how goals and tasks affect NFRs, which is helpful, e.g., for the selection of the most appropriate alternatives. In the absence of contribution links, as it is our case, *AwReqs* can be used to capture the fact that particular goals are important or even critical to meet NFRs and thus those goals' high rate of achievement is needed. This can be viewed as an operationalization of a contribution link, as we have just illustrated with *AR2*.

Preferable solutions

Alternatives introduced by OR-refinements are also frequently used to evaluate different means of satisfying a goal with respect to certain softgoals. In our approach, softgoals are refined into quality constraints and the qualitative contribution links are removed (and for our illustrations, we have not shown them at all).[3] However, the links do capture valuable information on the relative fitness of alternative ways to achieve goals. *AwReqs* can be used as a tool to make sure that "good" alternatives are still preferred over "bad" ones.

For instance, *AwReq AR9* states that schedules should be produced automatically by the system at least ten times more often than manually by the meeting organizer, presumably because this is faster or cheaper. This way the intuition behind softgoal contribution links is preserved. If multiple conflicting softgoals play roles in the selection of alternatives, then a number of alternative *AwReqs* can be created since the selection of the best alternative will be different depending on the relative priorities of the conflicting non-functional requirements.

[3]The rationale here is that softgoals do not have a clear-cut criteria for satisfaction and, therefore, cannot be referred to by *AwReqs*, i.e., their success or failure cannot be precisely determined.

Trade-offs

Conflicting NFRs (softgoals) usually impose some kind of trade-off in the system. For instance, when an *AwReq* targeting a quality constraint that refers to the system's performance fails, the system might switch to a more efficient solution, which in turn might result in a failure of another *AwReq*, concerned with the system's overall cost.

This kind of trade-off could be embodied in a single *AwReq*, if preferred, stating, e.g., that a given solution should fail *between 5 and 10 times a month* or that the success rate should be *between 80% and 90%*. The definition of both lower and upper bounds in these examples implicitly takes care of existing NFR trade-offs related to the requirement in question.

Preemptive adaptation

It is clear from our characterization of *AwReqs* that they are of reactive nature, i.e., once the system detects a situation that is not conformant with the specified requirements, it adapts. However, in many cases it could be better, or even necessary, to avoid the failure altogether, preemptively switching the system behavior.

A possible way of accomplishing this result using our approach is to have multiple *AwReqs* referring to the same requirement, but with different levels of criticality. For instance, in the Meeting Scheduler, *AR6* triggers adaptation actions when less than 75% of the meetings have *Good participation*. The value of 75% is the limit of tolerance given to failures of the quality constraint that operationalizes this softgoal. To prevent it from ever reaching this limit, other *AwReqs* establishing higher rates such as 80%, 90%, etc. could be modeled. Alternatively, we could use Trend *AwReqs* to detect when the success rate is decreasing, such as *AR14*.

Meta-*AwReqs*

A possible motivation for meta-*AwReqs* is the application of gradual reconciliation/compensations actions. This is the case with *AR13*: if *AR7* fails (i.e., *Participants use the system calendar* turns out to be false in a given occasion), one could think of a mild adaptation action, but if *AR3*'s success rate is actually less than 80% (i.e., the assumption is false at least two out of ten times), stronger action might be advised.

Another useful case for meta-*AwReqs* is to avoid executing specific reconciliation/compensation actions too many times. *AR13* can also illustrate this case: if the adaptation action associated with *AR7* is too costly, *AR13*'s adaptation could consist of disabling *AR7* (or changing its adaptation strategy) for a period of time.

Qualitative elicitation

One of the difficulties with *AwReqs* elicitation is coming up with precise specifications for the desired success rates over certain number of instances or during a certain time frame. To ease the elicitation and maintenance we recommend a gradual elicitation, first using high-level qualitative terms such as "medium" or "high" success rate, "large" or "medium" number of instances, etc. Thus, the *AwReq* may originate as "high success rate of R over medium number of instances" before becoming `SuccessRate(R, 95%, 500)`.

Of course, the quantification of these high-level terms is dependent on the domain and on the particular *AwReq*. So, "high success rate" may be mapped to 80% in one case and to 99.99% in another. Additionally, using abstract qualitative terms in the model while providing the mapping separately helps with the maintenance of the models since the model remains intact while only the mapping is changing.

5.2.2 Parameter identification

Input: partial specification of system behavior $S = \{G, I\}$;

Output: partial specification of system behavior $S = \{G, I, P\}$.

In this step, the requirements engineer should identify possible variations in the goal model affecting the indicators, which, therefore, can be manipulated to adjust the performance of the system. As described back in Section 4.1 (p. 91), these are captured by *control variables* and *variation points*.

For each indicator, the analyst should try to identify, again using existing elicitation techniques, if the model already shows variation points that could be exploited for the improvement of that indicator or if it is possible to create new variation points for this purpose. Existing works on requirements variability (cf. Section 2.1.4, p. 36) could be applied here.

In Section 4.1, we have also explained how control variables are abstractions over large or repetitive variation points. When identifying variability in requirements that could be exploited for adaptation, the analysts should decide between representing them using variation points or control variables, depending on the size and complexity of the model.

Requirement engineers should try to elicit at least one parameter for each indicator, if possible, in order to be able to use reconfiguration. Therefore, after at least one indicator has been analyzed, for each subsequent indicator one should check if the parameters that have already been identified may also be used to reconfigure to failures of the indicator at hand. The precise effect the parameter has on the indicator is, however, identified in the next step.

5.2.3 Relation identification

Input: partial specification of system behavior $S = \{G, I, P\}$;

Output: partial specification of system behavior $S = \{G, I, P, R'(I, P)\}$, where R' represents an unrefined set of relations.

In the third step of the process, the requirements engineer should identify the nature of the relations among the parameters identified in the previous step and the indicators (*AwReqs*) elicited in the first step. As seen in Section 4.1.2 (p. 94), such information is modeled in a qualitative way using differential relations.

There are two ways to perform a thorough identification of relations:

1. For each indicator from the set I the requirements engineer asks: which parameters from P does this indicator depend on?

2. Alternatively, iterate through set P and ask, for each parameter, which indicator in I is affected by it.

Either way, one should analyze all pairs $(i, p) \in \{I \times P\}$ and end up with a many-to-many association $R'(I, P)$ between these two sets. This set of relations R' is not yet final, as it needs to be refined in the final step of system identification.

To help analysts answer the questions proposed in the above enumeration, we provide some heuristics that may guide them in their analysis of the model and elicitation from stakeholders and domain experts:

- **Heuristic 1**: if provided in earlier steps of the Requirements Engineering process, softgoal contribution links capture these dependencies for *variation points*. The final specification for the Meeting Scheduler (cf. Figure 2.6, p. 37) had no such links, as they are not used at runtime in our approach, but earlier models could feature such links as, e.g., illustrated in Figure 5.2. The choices in *VP2* contribute to the softgoal *Low cost* and thus `VP2` affects the success rate of *Meetings cost less than €100*, a quality constraint (QC) derived from that softgoal. Any *AwReq*-derived indicator involving that QC is therefore also affected.

- **Heuristic 2:** another heuristic for deriving *potential* parameter-indicator relations is to link indicators to parameters that appear in the subtrees of the nodes the indicators are associated with. The rationale for this is the fact that parameters in a subtree rooted at some goal G, which models how G is achieved, change the subtree, thus potentially affecting the indicators associated with the goal. For instance, the parameter RfM is below the goal *Find a suitable room* in the tree and thus can

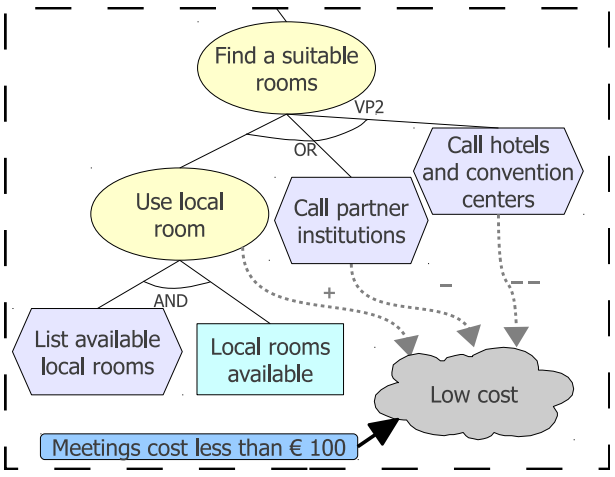


Figure 5.2: Contribution links may help identifying differential relations in variation points.

be (and actually *is*) affecting its success rate, an indicator (precisely, *AR4* — see Equation (4.20), p. 100).

- **Heuristic 3**: yet another way to identify potential parameter-indicator relations is to look at the non-functional concerns that these parameters/indicators address and to match the ones with the same concern. Earlier in Section 5.2.1 we described how non-functional requirements (NFRs) such as robustness, criticality, etc. lead to the introduction of *AwReqs* into goal models. The already-mentioned softgoal contributions explicitly link *variation points* with NFRs. Similar analysis should be done for *control variables*.

Further heuristics could be devised from more experiments in applying our approach to the design of adaptive systems.

5.2.4 Relation refinement

Input: partial specification of system behavior $S = \{G, I, P, R'(I, P)\}$;

Output: final specification of system behavior $S = \{G, I, P, R(I, P)\}$.

As briefly mentioned back in Section 4.2.2 (p. 105), the initial set of parameter-indicator relations produced in the previous step should be refined by comparing and combining those that refer to the same indicator. The process of refining the initial set of relations R' produces the final set of relations R, which is part of the final output of the system identification process.

The refinement step consists of separating the differential relations in R' in subsets $R_{i_1}, R_{i_2}, \ldots, R_{i_n}$, where $i_1, \ldots, i_n \in I$ and R_{i_k} contains the relations that affect indicator

i_k. Then, for each subset R_{i_k} with at least two elements, determine if the parameters related to indicator i_k can be (a) placed in order (total or partial); and (b) combined for cummulative effect.

In the Meeting Scheduler, for example (cf. Table 4.1, p. 100), when comparing two relations, say $\Delta(AR2/RfM) < 0$ and $\Delta(AR2/VP2) < 0$ (where $AR2$ refers to the success of quality constraint *Meetings cost less than €100*), the modeler can investigate whether either of these adaptation strategies is more effective than the other and by how much. This may result in the model being refined into, e.g., $|\Delta(AR2/RfM)| < |\Delta(AR2/VP2)|$, which would help the adaptation framework facing the choice between these alternatives. The analysis of whether selecting an alternative makes the value of an indicator match its reference input is to be addressed in future work.

The identification of cummulative effect, as also mentioned in Section 4.2.2, concerns the assumption that homogeneous impact is additive, i.e., if both p_1 and p_2 have a positive effect towards indicator i when increased, should we assume the default behavior in which changing both of them also produces a (possibly greater) positive effect — in other words, $\Delta(i/\{p_1, p_2\}) > 0$? An analogous question can be formulated for cummulative negative effect.

If this assumption is incorrect, a differential relation stating otherwise should be provided, as it is the case of RfM and $VP2$ with respect to $AR8$: $|\Delta(AR8/\{RfM, VP2\})| = |\Delta(AR8/VP2)|$ (cf. Equation (4.46), p. 100). In that case, changing both RfM and $VP2$ has the same effect as changing only $VP2$.

This example illustrates that when combining relations to analyze alternatives, care must be taken to only look at the parameters/indicators relevant in the current system configuration. Another example of this from the Meeting Scheduler is the parameter *View Private Appointments* (VPA), which cannot affect any indicator if the value of $VP1$ is not *Collect automatically*.

5.3 Adaptation Strategy Specification

Given the final specification of system behavior $S = \{G, I, P, R(I, P)\}$, developers would already have enough knowledge of the system in order to devise adaptation mechanisms using reconfiguration. Back in Section 4.2 (p. 99), we briefly mentioned one possible such mechanism:

1. Monitor all indicators $i \in I$. When an indicator i_k failure is detected, move to the next step;

2. Separate the relations subset $R_{i_k} \subset R(I, P)$, containing the relations that model the

effect of a parameter change on the indicator i_k that failed;

3. Considering all relations $r_{i_k} \in R_{i_k}$ use some criteria to select one of them. Possible criteria could be:

   - Select the one with the greatest positive effect on i_k, if the subset R_{i_k} is ordered;
   - Select the one with the least side-effects on other indicators. Given a relation, e.g., $\Delta(i_k/p_j) > 0$, its list of side-effects are all relations that contain p_j with opposite direction, e.g., $\Delta(i_x/p_j) < 0$;
   - If none of the above criteria can be applied or there is a tie, randomly select a relation.

4. Having selected a relation r_{i_k}, increase or decrease its associated parameter, depending on the nature of the relation. Use the smallest possible increment of the chosen parameter.

Although the above procedure might work in many cases, it is not given that it will always be the best course of action for any system failure. Section 4.2 provided a high-level description of many different algorithms that could be applied for reconfiguration (e.g., randomly picking the parameter, considering their effect on the indicator, oscillating the value of the parameter, applying a PID-like algorithm, etc.). Each different failure could benefit from a different kind of adaptation, that could even be using a procedure that is not described in this thesis, as our models are extensible.

Moreover, Section 3.2 (p. 78) shows that adaptation could consist not only of reconfiguration algorithms but also *Evolution Requirements* (*EvoReqs*), which change the requirements model itself in order to adapt. As previously mentioned, *EvoReqs* change the problem space, whereas reconfiguration algorithms work in the solution space only.

Both types of adaptation, however, are unified into a single concept of *adaptation strategy* (cf. sections 3.2.2 and 3.2.3, p. 83) that is associated with *AwReq* failures and specify what the system should do to adapt, closing the feedback loop. Such information can be used by developers in subsequent stages of the software development process (architectural design, coding) to implement the system, but also by a framework that implements the general behavior of the feedback loop, presented in Chapter 6.

Independently if they consist of reconfiguration or evolution, adaptation strategies are domain/application-dependent and, thus, have to be elicited from stakeholders and domain experts in order to best reflect their needs, associating the best adaptation to each failure. Therefore, after identifying information on the behavior of the system during system identification, requirements engineers should conduct a more targeted elicitation

process with the purpose of associating specific adaptation strategies to each of the possible *AwReq* failures of the system.

Later in Chapter 7 it will be seen that the experiments using our approach consisted of simulations of real-world failures with the purpose of verifying the response of our proposed framework to a few undesired situations at runtime. Unfortunately, we have not conducted any experiments with practitioners to evaluate the models and the proposed systematic process at design-time (a task which nonetheless remains in the list of future work).

For this reason, as with *AwReq* elicitation, we do not propose any particular technique for discovering which adaptation strategies to associate with each *AwReq*. Requirements engineers should consider this a process of elicitation of requirements for the adaptive capabilities of the system and use existing RE techniques for this purpose.

5.4 Chapter summary

In this chapter, we presented an overview of the *Zanshin* approach for the design of adaptive systems (§ 5.1), detailing its two main activities: *System Identification* (§ 5.2) and *Adaptation Strategy Specification* (§ 5.3). These activities are conducted in parallel, possibly in an iterative fashion, with "vanilla" Requirements Engineering activities in order to engineer the requirements for adaptation of the system-to-be.

During System Identification, requirements engineers should identify the important indicators that should be monitored (§ 5.2.1), the parameters that can have an effect on such indicators when changed (§ 5.2.2) and the relations between each *indicator*/*parameter* pair (§ 5.2.3). Finally, such relations can be refined if further information about them is available (§ 5.2.4).

The second step, Adaptation Strategy Selection, consists simply in assigning to each of the identified indicators a list of adaptation strategies that represent the requirements for adaptation elicited from the stakeholders. Chapters 3 and 4 described how these requirements can be modeled and we do not propose any particular technique for their elicitation.

Chapter 6

Architectural considerations: the *Zanshin* framework

Talk is cheap. Show me the code.

Linus Torvalds

In the previous chapter, we described the steps to produce models that represent the requirements for system adaptation. This process partially addresses research question **RQ3** (cf. Chapter 1), which stated: *How can we help architectural designers and programmers implement this requirements-based feedback loop?* The models described in chapters 3 and 4 produced by the process presented in Chapter 5 can guide developers in their task of designing the system architecture and implementing it in code.

However, if our premise is that adaptation will be operationalized through a feedback loop architecture (cf. Figure 2.8, p. 41), the generic features of monitoring for requirements divergence, parsing the requirements specification, deciding the most appropriate adaptation strategy to apply to a given failure and informing the target system of the selected strategy at runtime can be implemented into a generic framework that can be reused by developers, reducing the amount of work in architectural design and coding.

We have, therefore, implemented a prototype for such a framework which, as the systematic process presented earlier, is called *Zanshin*. Its source code is available in a public version control repository located at `http://github.com/vitorsouza/Zanshin` and, in this chapter, we provide an overall description of the framework and detail each one of its components.

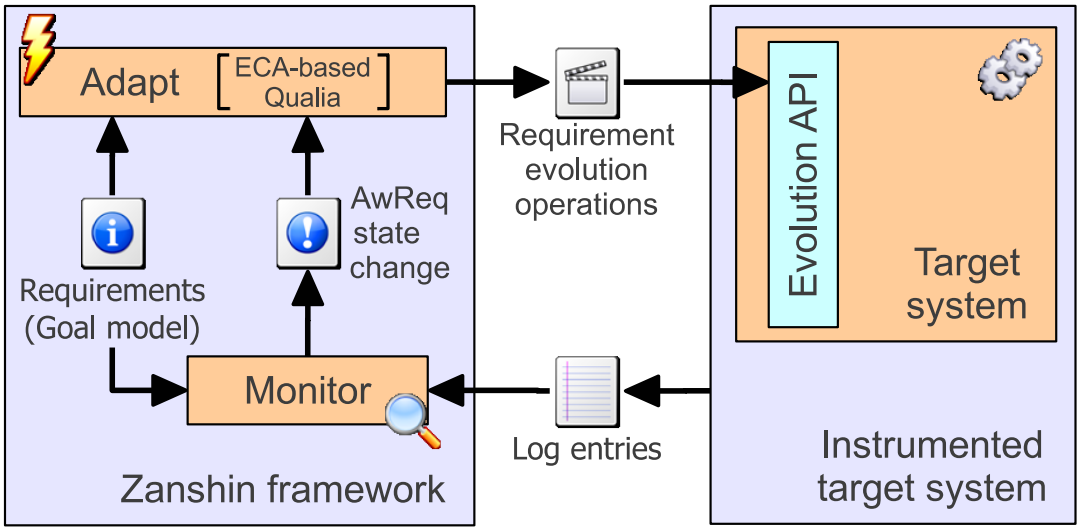


Figure 6.1: Overview of the *Zanshin* framework.

6.1 Overview

Figure 6.1 gives an overview of the *Zanshin* framework. First of all, the *Target system* (the system to which the framework will add adaptation capabilities) is instrumented in order to provide a log that indicates when instances of requirements (i.e., elements of the goal model as explained back in Section 2.1.3, p. 32) have changed state (considering the states presented back in Figure 3.1, p. 66).

Given this log and the requirements specification — a goal model, as before, but with Awareness Requirements (*AwReqs*) added (cf. Section 3.1, p. 64) —, the *Monitor* component is able to conclude if and when certain *AwReqs* have themselves changed state (which includes not only *AwReq* failures, but also *AwReqs* being satisfied).

These state changes should then trigger an *Adapt* component that decides which requirement evolution operations the target system should execute (cf. Section 3.2, p. 78). This component can be divided in two main parts:

- An Event-Condition-Action (ECA) based adaptation component that chooses an adaptation strategy based on the list of strategies associated with the *AwReq* failure and their respective applicability conditions;

- A qualitative reconfiguration component (called *Qualia*), which is activated by the ECA-based process when reconfiguration is selected as the appropriate strategy to apply, that executes the reconfiguration algorithm (cf. Section 4.2, p. 99) that has been specified in the adaptation strategy.

In either case, the output of the *Adapt* component is a list of Evolution Requirement (*EvoReq*) operations, as the ones shown back in Table 3.5 (p. 82) that are sent to the *Target system*. The latter, in its turn, should carry on application and domain-specific actions based on the instructions given by the *EvoReq* operations. In Figure 6.1, this is represented by the *Evolution API* component, which should be implemented by the *Target system*.

6.1.1 Implementation

The *Zanshin* framework was implemented as six OSGi[1] bundles (components): Core, Logging, Monitoring, Adaptation, *Qualia* and Simulation. The Core bundle exposes five service interfaces, based on the framework's architecture shown in Figure 6.1, each of which implemented by a different bundle:

- *Monitoring Service*: monitors the log provided by the target system and detects changes of state in *AwReq* instances, submitting these to the Adaptation Service;
- *Adaptation Service*: implements the aforementioned ECA-based adaptation process, analyzing the requirements specification and deciding which adaptation strategy to execute next;
- *Reconfiguration Service*: implemented by *Qualia*, executes the aforementioned reconfiguration algorithm that has been specified in the adaptation strategy;
- *Target System Controller Service*: implemented by the Simulation bundle, serves as a bridge between the adaptation framework and the target system, by implementing the *EvoReq* operations that are called by the executed adaptation strategies;
- *Repository Service*: implemented by the Core bundle itself, stores the instances of the requirements models that are used by the other services.

Requirements models are specified using Eclipse Modeling Framework (EMF)[2] meta-models: the Core component provides the basic GORE classes (cf. Figure 3.2, p. 71) and the classes involved in the ECA-based process (presented later in Section 6.3). These

[1] The Open Services Gateway initiative framework is a module system and service platform for the Java™ programming language. It allows components to be implemented as bundles which can be remotely installed, started, stopped, updated, and uninstalled without requiring the component container itself to be restarted. See `http://www.osgi.org/`.

[2] EMF is a modeling framework an code generation facility for the Eclipse platform. From a model described in an XML-based language, EMF can produce a set of Java classes representing the model, as well as adapters and editors. See `http://www.eclipse.org/modeling/emf/`.

meta-models are extended by the Simulation bundle to provide classes representing the requirements of the target system.

For example, for the Meeting Scheduler there would be one EMF class for each requirement of its goal model, shown a few times in previous chapters (e.g., Figure 4.1, p. 92), extending the appropriate GORE/ECA classes (see also the list of their UML representation in Table 3.2, p. 72).

Finally, the target system' requirements specification can be written as an EMF model, to be read by the framework, represented in memory as Java™ objects (using EMF's API) and stored in the Repository Service when the target system is executed. This way, the EMF model represents the requirements at the *class* level, whereas the objects stored in the Repository Service for each execution represent the requirements at the *instance* level. Listing 6.1 shows parts of the specification of the Meeting Scheduler requirements in EMF.

Listing 6.1: A section of the Meeting Scheduler requirements specification in EMF.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <scheduler:SchedulerModel ...>
3   <!-- The "vanilla" goal graph. -->
4   <rootGoal xsi:type="scheduler:G_SchedMeet">
5    <children xsi:type="scheduler:T_CharactMeet"/>
6    <children xsi:type="scheduler:G_CollectTime">
7     <children xsi:type="scheduler:T_CallPartic"/>
8     <children xsi:type="scheduler:T_EnaukOartuc"/>
9     <children xsi:type="scheduler:G_CollectAuto">
10     <children xsi:type="scheduler:D_ParticUseCal"/>
11     <children xsi:type="scheduler:T_CollectCal"/>
12    </children>
13    ...
14   </children>
15   ...
16
17   <!-- Awareness Requirements. Starting at //@rootGoal/@children.6. -->
18   <children xsi:type="scheduler:AR1" target="//@rootGoal/@children.0"/>
19   ...
20  </rootGoal>
21
22  <!-- System parameters. -->
23  <configuration>
24   <parameters xsi:type="scheduler:CV_RF" type="ecv"/>
25   ...
26  </configuration>
27
28  <!-- Indicator / parameter differential relations. -->
29  <relations indicator="//@rootGoal/@children.6" parameter="//@configuration/
        @parameters.0" operator="ft" />
30  ...
31 </scheduler:SchedulerModel>
```

The first part of the specification (lines 3–14) shows the elements of the Meeting Scheduler's goal model organized in the same tree-like structure used to present them graphically in a goal model diagram. Further down, in line 18, *AwReq AR1* is included, specifying as its target the element of index `0` in the `children` set of the element specified by the tag `<rootGoal />`, i.e., `T_CharactMeet`, which represents task *Characterize meeting*.

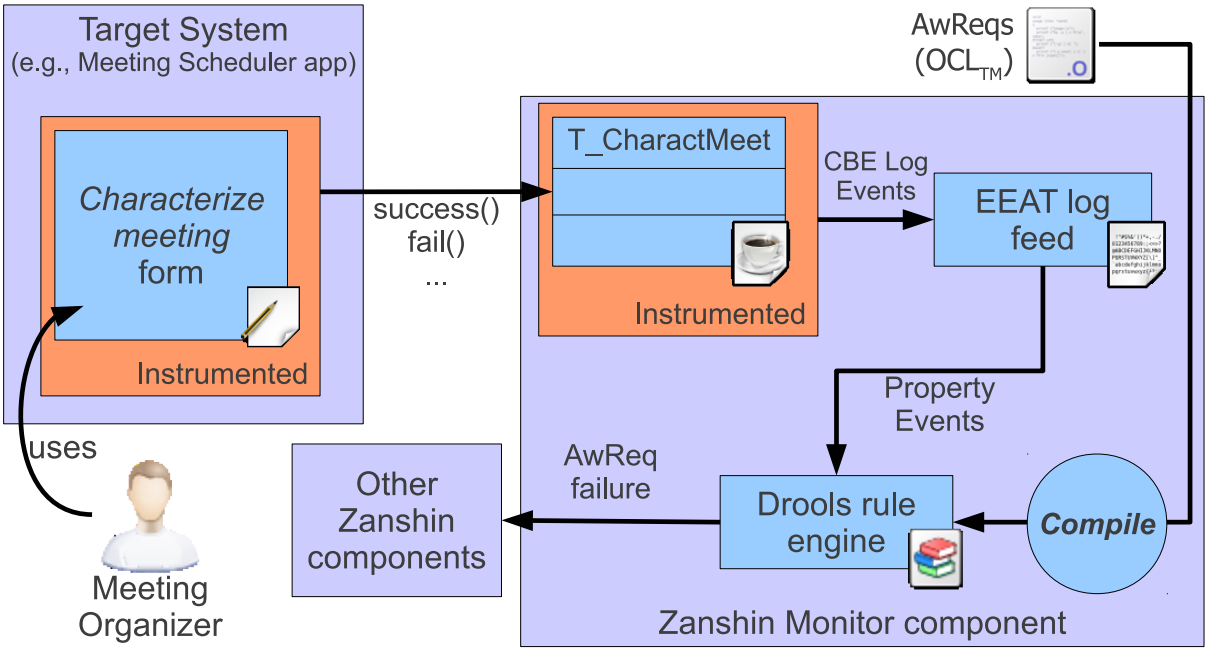


Figure 6.2: Overview of *Zanshin*'s Monitor component.

The `<configuration>` tag encloses the parameters elicited during system identification and Listing 6.1 shows parameter *RF* (*Required Fields*) as an example (line 24). Finally, the goal model contains a set of `relations`, such as the one in line 29: the `indicator` is *AR1* (`//@rootGoal/@children.6`), the parameter is *RF* (`//@configuration/@parameters.0`) and the relation is "fewer than" (`ft`). Thus, the example illustrates differential relation $\Delta(AR2/RfM) < 0$, shown earlier in Equation (4.16) (p. 100).

The next sections explain in more detail how the main components of the *Zanshin* framework work, namely, the *Monitor* component, the *ECA-based adaptation* component and *Qualia*, the *Qualitative reconfiguration* component.

6.2 The monitor component

Figure 6.2 shows an overview of *Zanshin*'s Monitor component. On the left-hand side, the target system is represented, exemplified by the Meeting Scheduler application (implemented in whatever platform was chosen during the architectural design of the system). For monitoring to work, the source code of the monitored system (in this case, the Meeting Scheduler) has to be instrumented in order to create the instances of the classes that represent the requirements at runtime (cf. Table 3.2, p. 72) and call methods defined in classes `DefinableRequirement` and `PerformativeRequirement` (cf. Figure 3.2, p. 71), namely:

- `start()`: when the user starts executing a task, this method is called in the instance that represents that task. Then, *Zanshin* propagates it up the goal tree, calling the same method in instances representing ancestor goals that have not yet been started. For quality constraints and domain assumptions, this method should be called immediately before their satisfaction is to be evaluated;

- `success()`: this method is called when a requirement has been satisfied. As before, *Zanshin* will propagate up the goal tree the satisfaction of ancestor goals, according to the Boolean semantics specified in their refinements (cf. Section 2.1.3, p. 32);

- `fail()`: this method is called when a requirement has not been satisfied. Like the `success()` method, *Zanshin* also propagates failure up the goal tree;

- `cancel()`: for long-running, performative requirements (such as goals and tasks), this method is called when the requirement has been canceled by the user. Cancellation is also propagated up the tree, like `success()` and `fail()` calls;

- `end()`: this method is called automatically by *Zanshin* after one of the three possible final outcomes for a requirement: success, fail or cancel.

As previously mentioned, the classes that represent the requirements at runtime belong to the *Zanshin* framework, depicted in the right-hand side of the figure. As explained back in Section 2.1.6 (p. 43), we have used EEAT to monitor *AwReqs* and, thus, instrumentation is also used in the framework itself in order to send *log events* in the Common Base Event (CBE)[3] format to *EEAT's log feed*, which transforms the log entries into *property events* before sending them to the *Drools rule engine*.[4]

In order to detect *AwReq* failures (or any other change of state), the rule engine needs to receive as input the definition of each *AwReq*, compiled to its rule specification language. EEAT provides a compiler that can derive such rules from *AwReqs* expressed in OCL_{TM}. However, EEAT cannot generate proper rules from the human-friendly OCL_{TM} specifications illustrated in Chapter 3, such as the ones illustrated for the meeting scheduler in Listing 3.1 (p. 73). To explain why, let us recall one simple *AwReq* definition, shown in Listing 6.2, below.

Listing 6.2: A simple *AwReq*, represented in human-friendly OCL_{TM}.

```
1 -- AwReq AR1: task 'Characterize meeting' should never fail (human-friendly
      version).
2 context T_CharactMeet
3   inv AR1: never(self.oclInState(Failed))
```

[3] See `http://www.ibm.com/developerworks/library/specification/ws-cbe/`.

[4] Drools is a production rule system that provides a forward chaining inference based rules engine. See `http://www.jboss.org/drools`.

AR1 indicates that no instance of task *Characterize meeting* should ever be in the `Failed` state. However, this specification is unbounded in time: the rule engine will never conclude that this invariant has been satisfied because there is always the possibility that one instance will be in the `Failed` state sometime in the future. On the other hand, if an instance actually switches to the `Failed` state, the invariant is violated and will stay that way forever for this same reason.

Therefore, in order to be able to verify this constraint for every instance and to determine its satisfiability in any case, we have to transform the initial, human-friendly specification of the *AwReqs* to one, EEAT-ready specification, which is based on the aforementioned methods received by the run-time instances that represent the requirements. The EEAT-ready version of *AR1* is shown in Listing 6.3.

Listing 6.3: Same *AwReq* from Listing 6.2, represented in EEAT-ready OCL_{TM}.

```
1 -- AwReq AR1: task 'Characterize meeting' should never fail (EEAT-ready version)
      .
2 context T_CharactMeet
3   inv AR1: between(receivedMessage('start') <> null, receivedMessage('end') <>
        null, never(receivedMessage('fail') <> null))
```

Together with the `between` clause (one of Dwyer et al. [1999] scopes, cf. Section 2.1.6, p. 43), these methods allow us to define the period in which *AwReqs* should be evaluated, because otherwise the rule system could wait indefinitely for a given message to arrive.

Given the right scope, the methods `success()`, `fail()` and `cancel()` are called by the monitored system to indicate a change of state in the requirement from `Undecided` to one of the corresponding final states (cf. Figure 3.1, p. 66). These methods are then used in the EEAT-ready specification of *AwReqs*. Therefore, in practice, we define *AR1* not as never being in the `Failed` state, but as never receiving the `fail()` message in the scope of a single execution (between `start()` and `end()`).

An aggregate requirement, on the other hand, aggregates the calls during the period of time defined in the *AwReq*, as shown in Listing 6.4. For *AR6*, this is done by monitoring for calls of the `newMonth()` method, which are called automatically by the monitoring framework at the beginning of every month. Similar methods for different time periods, such as `newDay()`, `newHour()` and so forth, should also be implemented.

Listing 6.4: The EEAT-ready version of an aggregate *AwReq*.

```
1 -- AwReq AR6: QC 'At least 90% of participants attend' should have a 75% success
      rate per month (EEAT-ready version).
2 context Q_Min90pctPart
3   def: beg : LTL::OclMessage = receivedMessage('newMonth')
4   def: end : LTL::OclMessage = receivedMessage('newMonth')
5   def: wS : Integer = receivedMessages('success')->select(x | new Date().
        difference(x.time, MONTH) == 1)->size()
6   def: wF : Integer = receivedMessages('fail')->select(x | new Date().difference
        (x.time, MONTH) == 1)->size()
7   inv AR6: between(weekA <> null, weekB <> null and weekA.date().difference(
        weekB.date(), MONTH) == 1, always(wS / (wS + wF) >= 0.75)
```

Table 6.1: EEAT/OCL$_{TM}$ idioms for some *AwReq* patterns.

Pattern	**OCL$_{TM}$ idiom**
NeverFail(R)	`def: rm: OclMessage = receiveMessage('fail')` `inv pR: never(rm)`

SuccessRate(R, r, t)	`def: msgs: Sequence(OclMessage) = receiveMessages()->`
	`  select(range().includes(timestamp()))`
	`-- Note: these definitions are patterns that are assumed in`
	`  the following definitions`
	`def: succeed: Integer = msgs->select(methodName = 'succeed'))->size()`
	`def: fail: Integer = msgs->select(methodName = 'fail'))->size()`
	`inv pR: always(succeed / (succeed + fail) > r)`

ComparableSuccess (R, S, x, t)	`-- c1 and c2 are fully specified class names` `inv pR: always(c1.succeed > c2.succeed * x)`
MaxFailure(R, x, t)	`inv pR: always(fail < x)`
P_1 and/or P_2; not P	`-- arbitrary temporal and real-time logical expressions are` `  allowed over requirements definitions and run-time objects`

An automatic translator from the *AwReqs*' initial specification to their EEAT-ready specification could be built to aid the designer in this task. Another possibility is to go directly from the *AwReq* patterns (cf. Section 3.1.3, p. 75) to this final specification. Table 6.1 illustrates how some of the patterns of Table 3.3 (p. 76) can be expressed in OCL$_{TM}$.

These formulations are consistent with those shown in listings 6.3 and 6.4. The definitions and invariants are placed in the context of the UML classes that represent requirements at runtime. For example, a `receiveMessage('fail')` for context `R`, denotes the called operation `R.fail()` for class `R`. Therefore, the invariant `pR` in the first row of table 6.1 is true if `R.fail()` is never called.

Of course, the patterns of Table 3.3 represent only common kinds of expressions. *AwReqs* contain the range of expressions where a requirement R_1 can express properties about requirement R_2, which include both design-time and run-time requirements properties. OCL$_{TM}$ explicitly supports such references, as the expressions in Listing 6.5 illustrate:

Listing 6.5: Generic property evaluation in OCL$_{TM}$, supported by EEAT.

```
1 def: p1: PropertyEvent = receivedProperty('p:package.class.invariant')
2 inv p2: never(p1.satisfied() = false)
```

In OCL$_{TM}$, all property evaluations are asserted into the run-time evaluation repository as `PropertyEvent` objects. The definition expression of `p1` refers to an invariant

(on a UML class, in a UML package). Properties about `p1` include its run-time evaluation (`satisfied()`), as well as its design-time properties (e.g., `p1.name()`). Therefore, in OCL_{TM}, requirements can refer to their design-time and run-time properties and, thus, *AwReqs* can be represented in OCL_{TM}.

To help developers experiment with EEAT-ready specifications of *AwReqs* in OCL_{TM}, in order to verify if they are correct, we have developed a front-end graphical tool that shows the information being received by EEAT, the methods being called in the run-time instances of the requirements and, finally, the `PropertyEvent` objects generated by EEAT. A screen shot of this tool is shown in Figure 6.3.

6.3 The ECA-based adaptation component

Referring back to Figure 6.2, once the monitoring component detects a change in the state of an *AwReq* (i.e., that an *AwReq* has been satisfied or failed), it sends a notification to the *Adapt* component. This component uses an ECA-based process to execute adaptation strategies in response to system failures. This process goes through the list of strategies associated with the failed *AwReq*, selecting and executing the most appropriate one based on some conditions. See, for instance, Table 3.7 (p. 87) for the strategies associated with *AwReqs* of the Meeting Scheduler.

This process is summarized in the algorithm shown in Listing 6.6, which manipulates instances of the classes represented in the class model of Figure 6.4.

Listing 6.6: ECA-based algorithm for responding to *AwReq* failures.

```
1  processEvent(ar : AwReq) {
2   session = findOrCreateSession(ar.class);
3   session.addEvent(ar);
4   solved = ar.condition.evaluate(session);
5   if (solved) break;
6
7   ar.selectedStrategy = null;
8   for each s in ar.strategies {
9     appl = s.condition.evaluate(session);
10    if (appl) {
11      ar.selectedStrategy = s;
12      break;
13    }
14  }
15
16  if (ar.selectedStrategy == null)
17    ar.selectedStrategy = ABORT;
18
19  ar.selectedStrategy.execute(session);
20  ar.condition.evaluate(session);
21 }
```

The process is triggered by *AwReq* evaluations, independent of the *AwReq* instance's final state (`Success`, `Failed` or `Canceled`). For instance, let us recall one of the *AwReqs* of the Meeting Scheduler (cf. Table 3.7): say the weekly success rate of *Collect timetables*

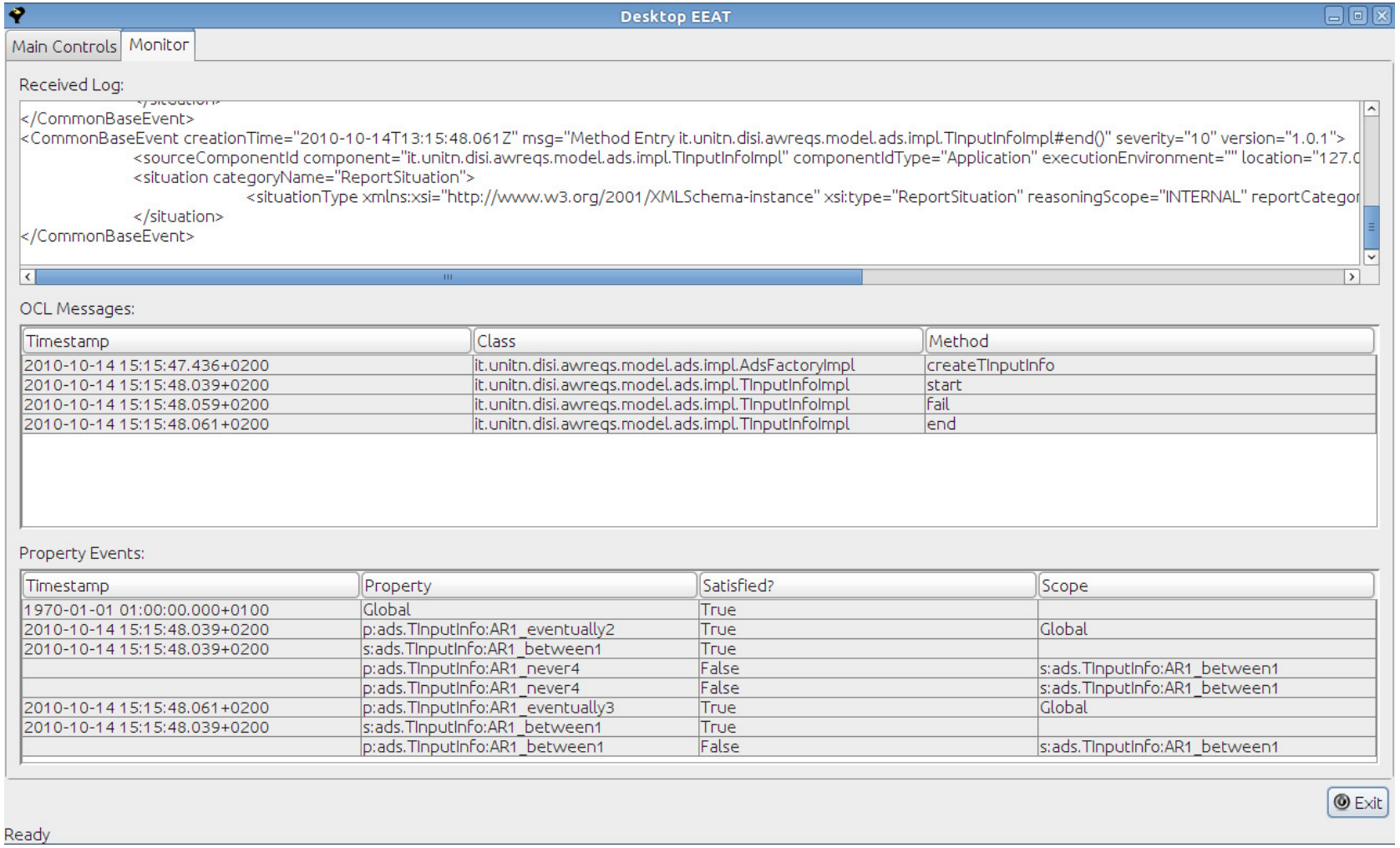


Figure 6.3: Graphical front-end for monitoring the satisfaction of *AwReqs* using EEAT's infrastructure.

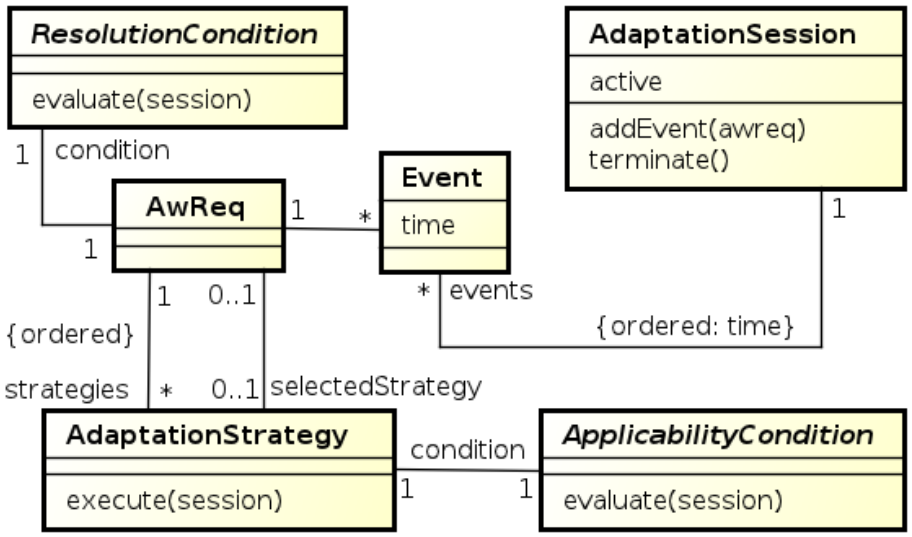


Figure 6.4: Entities involved in the ECA-based adaptation process.

has decreased twice in a row, causing the failure of *AR3* and starting the ECA process.

The algorithm begins by obtaining the *adaptation session* that corresponds to the class of said *AwReq*, creating a new one if needed (line 2). As shown in Figure 6.4, an *adaptation session* consists on a series of events, referring to *AwReq* evaluations. This time-line of events can be later used to check if a strategy is applicable or if the problem has been solved (i.e., if the adaptation has been successful). Active sessions are stored in a repository (e.g., a hash table indexed by *AwReq* classes attached to the user session) which is managed by the `findOrCreateSession()` procedure. In the example, assuming it is the first time *AR3* fails, a new session will be created for it.

Then, the process adds the current *AwReq*'s evaluation as an event to the active session, immediately evaluates if the problem has been solved — this is done by considering the *AwReq*'s *resolution condition*, which analyzes the session's event time-line — and stops the process if the answer is affirmative (3–5). For example, the trivial case is considering the problem solved if the (next) *AwReq* evaluates to success, but this abstract class can be extended to provide different kinds of *resolution conditions*, including, e.g., involving a human-in-the-loop to confirm if the problem has indeed been solved, organizing conditions into AND/OR-refinement trees (like in a goal model), etc. For the running example, let us say that *AR3* has been associated with the aforementioned simple resolution condition. Since the *AwReq*'s state is *Failed*, the session is not considered solved and the algorithm continues.

If the current *AwReq* evaluation does not solve the issue, the process continues to search for an applicable *adaptation strategy* to execute in order to try and solve it (7–14). It does

so by going through the list of strategies associated with the *AwReq* that failed in their predefined order (e.g., preference order established by the stakeholders) and evaluating their *applicability conditions*, breaking from the loop once an applicable strategy has been found. As with `ResolutionCondition`, `ApplicabilityCondition` is also abstract and should be extended to provide specific kinds of evaluations. For instance, apply a strategy "at most N times per session/time period", "at most in X% of the failures/executions", "only during specified periods of the day", AND/OR-refinements, etc. (patterns can be useful here). Some conditions might even need to refer to some domain-specific properties or contextual information. If no applicable strategy is found, the process falls back to the *Abort* strategy (16–17).

Back to the running example, imagine now that the Meeting Scheduler designers have associated two strategies to *AR3*. First, relax it by replacing *AR3* with *AR3_3weeks*, which verifies if the success rate has decreased not in two, but in three consecutive weeks (i.e., `not TrendDecrease(G_CollectTime, 7d, 3)`). This strategy is associated with a condition that constraints its applicability to at most once a trimester. Second, the *Warning* strategy is also associated with *AR3*, sending a message to the IT support staff so they can take corrective action. To this strategy a simple applicability condition is associated, which always returns true. Therefore, if this is the first time *AR3* fails in the past three months, it will be relaxed to *AR3_3weeks*, otherwise the *Warning* strategy will be selected.

After the strategy is selected, it is executed and the session is given another chance to evaluate its resolution (sometimes we would like to consider the issue solved after applying a specific strategy, independent of future *AwReq* evaluations, e.g. when we use *Abort*). When an *adaptation session* is considered resolved, it should be *terminated*, which marks it as no longer being active. At this point, future *AwReq* evaluations would compose new *adaptation sessions*. Instead, if the algorithm ends without solving the problem, the framework will continue to work on it when it receives another *AwReq* evaluation and retrieves the same *adaptation session*, which is still active. Some *adaptation strategies* can force a re-evaluation of the *AwReq* when executed, which guarantees the continuity of the adaptation process.

For the *AR3* example, the session would remain active until another month has been passed and *AR3_3weeks* is checked. If the success rate increases then, *AR3_3weeks* will be satisfied, triggering another call to `processEvent()`, which would find *AR3*'s session and, according to the resolution condition, consider it solved and terminate it. If the rate decreases one more time, though, the *Warning* strategy is used and the session remains active until the following week.

As this example illustrated, information on resolution and applicability conditions

should be present in the requirements specification in order for the adaptation framework to use this process. We do not propose any particular syntax for the inclusion of this information in the specification. Listing 6.7 demonstrates how *AwReqs*' resolution conditions, adaptation strategies and their applicability conditions are specified in the EMF file that encodes the requirements specification, extending the section that was illustrated in the beginning of the chapter, in Listing 6.1.

Listing 6.7: EMF specification of *AR3*'s strategies and conditions.

```
<?xml version="1.0" encoding="UTF-8"?>
<scheduler:SchedulerModel ...>
 <!-- The "vanilla" goal graph. -->
 <rootGoal xsi:type="scheduler:G_SchedMeet">
  ...

  <!-- Awareness Requirements. -->
  ...
  <children xsi:type="scheduler:AR3" target="//@rootGoal/@children.1">
   <condition xsi:type="model:SimpleResolutionCondition"/>
   <strategies xsi:type="model:RelaxReplaceStrategy" newRequirement="//@rootGoal
       /@children.9">
    <condition xsi:type="model:MaxExecApplicabilityCondition" maxExecutions="1"/
        >
   </strategies>
   <strategies xsi:type="model:WarningStrategy" actor="//@actors.3" />
  </children>
  <children xsi:type="scheduler:AR3_3weeks" target="//@rootGoal/@children.1">
   ...
  </children>
  ...
 </rootGoal>
 ...
</scheduler:SchedulerModel>
```

Finally, it is important to note that the ECA-based process is only one possible solution for the coordination and execution of adaptation strategies in response to *AwReq* failures at runtime. It can be replaced or combined with other processes that use *EvoReqs* and any extra specification necessary (e.g. applicability and resolution conditions) to: (a) select the best strategy to apply; (b) execute it; (c) check if the problem has been solved; (d) loop back to the start if it has not. Being developed in OSGi bundles implementing well-defined services, *Zanshin* offers an extensible architecture that allows other adaptation services to be plugged-in and used.

6.4 The qualitative reconfiguration component

In Section 4.2 (p. 99), we have introduced the *Qualia* framework, which allowed us to specify reconfiguration algorithms with different levels of precision and associate them with *AwReq* failures.

As mentioned earlier, we have implemented *Qualia* as a *Zanshin* component (i.e., bundle) to take advantage of its infrastructure. A new adaptation strategy called *Reconfiguration Strategy* (with customized applicability and resolution conditions) was added

to the framework, as well as the capability of recognizing OSGi bundles that provide *Reconfiguration Services*, allowing for new reconfiguration algorithms to be plugged in the framework dynamically.

Qualia's bundle registers one such service, which executes the steps of its adaptation process intertwined with *Zanshin*'s ECA-based process, detailed in the previous subsection. The framework's EMF meta-models were also extended to allow the specification of the information required by *Qualia*'s adaptation procedures. Listing 6.8 shows an example from the Meeting Scheduler specification.

Listing 6.8: EMF specification of a reconfiguration strategy using *Qualia*.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <scheduler:SchedulerModel ...>
3   <!-- The "vanilla" goal graph. -->
4   <rootGoal xsi:type="scheduler:G_SchedMeet">
5    ...
6
7    <!-- Awareness Requirements. -->
8    ...
9    <children xsi:type="scheduler:AR6" target="//@rootGoal/@children.20/@children
         .0">
10    <condition xsi:type="model:ReconfigurationResolutionCondition"/>
11    <strategies xsi:type="model:ReconfigurationStrategy" algorithmId="qualia">
12     <condition xsi:type="model:ReconfigurationApplicabilityCondition"/>
13     <procedureIds xsi:type="ecore:EString">ordered-effect-parameter-choice</
           procedureIds>
14     <properties xsi:type="model:AlgorithmProperty" key="order" value="desc"/>
15     <properties xsi:type="model:AlgorithmProperty" key="n" value="2"/>
16    </strategies>
17   </children>
18    ...
19   </children>
20   ...
21  </rootGoal>
22  ...
23 </scheduler:SchedulerModel>
```

The `ReconfigurationStrategy` is associated with *AR6*'s specification, defining `qualia` as its algorithm (line 11). The *Reconfiguration Strategy* will then ask a factory class to provide the service associated with the `qualia` identifier, obtaining a reference to *Qualia*'s *Reconfiguration Service*. Special resolution and applicability conditions — namely, `ReconfigurationResolutionCondition` (10) and `ReconfigurationApplicabilityCondition` (12) — need to be used in order to intertwine *Zanshin*'s ECA-based process with *Qualia*'s algorithm. Both these classes can wrap other resolution or applicability conditions, respectively, allowing the developer to specify in *Qualia* any condition that is available in *Zanshin*.

Inside the strategy definition, the tag `<procedureIds />` can be used to determine procedures that will replace the default ones in order to form the desired reconfiguration algorithm. In the example (line 13), the default parameter choice procedure is replaced with {*Ordered Effect Parameter Choice*} procedure, as per the Meeting Scheduler's specification (cf. Table 3.7, p. 87).

Finally, the properties *[order = descending, n = 2]*, also associated with this adaptation strategy in the specifications, are represented in EMF using the `<properties />` tag (lines 14 and 15). *Qualia* reads these values and execute the specified reconfiguration algorithm when a failure of *AR6* is detected.

In the next chapter, we provide more examples of use of the *Zanshin* and *Qualia* frameworks in the context of our experiments with the design of an Adaptive Computer-aided Ambulance Dispatch system.

6.5 Performance evaluation

Other than the experiments to be presented next in Chapter 7, which focus on showing that *Zanshin* produces sensible responses to failures at runtime based on the augmented requirements specification, we have also conducted performance experiments to evaluate the scalability of the framework. The performances of the *Monitor* and *Adapt* framework were evaluated separately and the results are reported below.

6.5.1 Performance of the *Monitor* component

Monitoring has little impact on the target system, mostly because the target system and the monitor typically run on separate computers. The TPTP Probekit[5], used by EEAT to instrument the source code of the target system in order to provide the required log entries, provides optimized byte-code instrumentation, which adds little overhead to some (selected) method calls. The logging of significant events consumes no more than 5%, and typically less than 1% overhead.

For real-time monitoring, it is important to determine if the target events can overwhelm the monitoring system. A performance analysis of EEAT was conducted by comparing the total monitoring runtime vs. without monitoring using 40 combinations of the Dwyer et al. [1999] temporal patterns (cf. Section 2.1.6, p. 43). For data, a simple two-event sequence was the basis of the test datum; for context, consider the events as an arriving email and its subsequent reply. These pairs were continuously sent to the server 10 thousand times. In the experiment, the event generator and EEAT ran in the same multi-threaded process. The test ran as a JUnit[6] test case within Eclipse[7] on a Windows Server 2003 dual core 2.8 GHz with 1GB memory. The results suggest that, within the test configuration, sequential properties (of length 2) are processed at 137 event-pairs per

[5]See `http://www.eclipse.org/tptp/`.
[6]See `http://www.junit.org/`.
[7]See `http://www.eclipse.org/`.

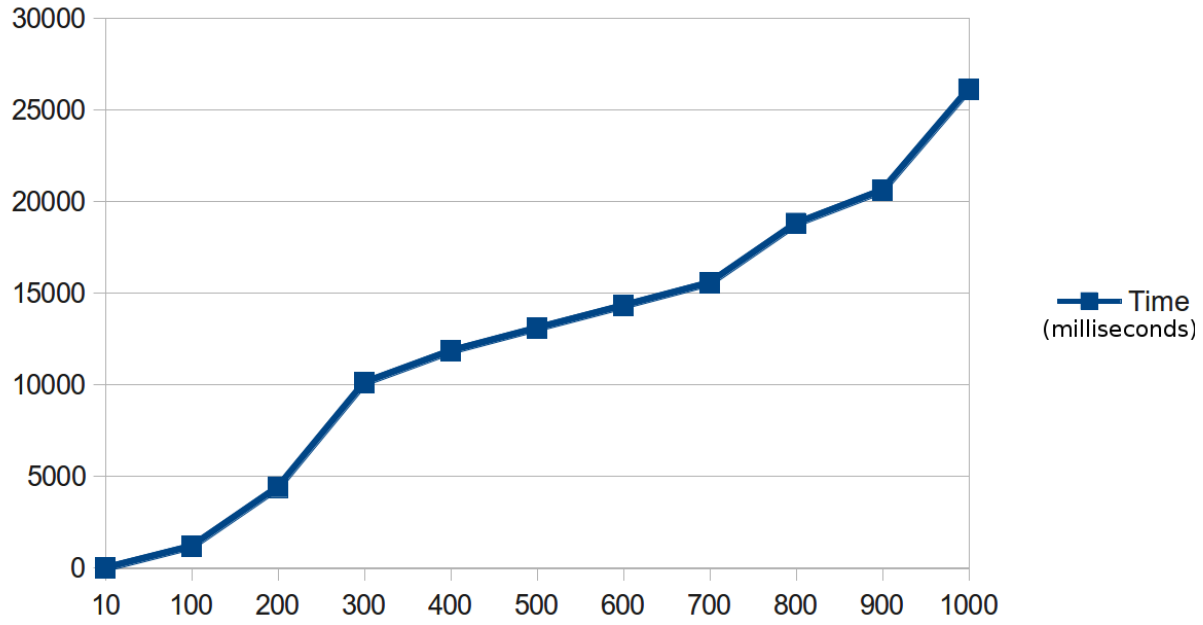


Figure 6.5: Results of the scalability tests of *Zanshin*.

second [Robinson and Fickas, 2009]. This indicates that EEAT is reasonably efficient for many monitoring problems.

6.5.2 Performance of the *Adapt* component

To evaluate the performance of *Zanshin*'s ECA-based adaptation process, we have developed a simulation in which goal models of different sizes (100–1000 elements) are built and have an *AwReq* failing at runtime. The framework applies the adaptation strategy that is also included in the specification and the target system (i.e., the simulation) acknowledges it. The simulation was ran ten times for each goal model size and the running time of the framework was calculated. Average times in milliseconds for each goal model size are shown in Figure 6.5 (the running time of the target system was irrelevant in comparison and, therefore, not included in the graph).

As the graph shows, the adaptation framework scales linearly with the size of the goal model. The interested reader can experiment the simulations for themselves by downloading the source code of the framework. Furthermore, as with the *Monitor* component, the target system and adaptation framework can be ran in a separate computers, reducing the impact of the adaptation process even further.

Another, similar, simulation uses a randomly generated goal model with different number of parameters (again, from 100 until 1000, scaling up by 100 elements each time), all of them related to a failing *AwReq*. *Zanshin* and *Qualia* were timed in ten sequential

executions of this simulation and average times for each number of parameters show the framework also scales linearly when *Qualia* is chosen as the reconfiguration service.

In effect, by analyzing *Qualia*'s default algorithm (cf. Section 4.2.1, p. 102), one can conclude that its complexity is $O(N \times R)$, where N is the *number of parameters* to choose and R is the number of differential relations in the model. With the proper data structures, however, this complexity can be further reduced.

6.6 Chapter summary

In this chapter, we described a prototype framework that has been created as part of our research in order to help developers with the implementation of adaptive systems, as the framework already contains the generic features of a feedback loop that operationalizes adaptation based on the models presented earlier, in chapters 3 and 4.

The framework, which as the process described in Chapter 5 is also called *Zanshin*, is composed of five different services: monitoring, adaptation, reconfiguration, controller and repository (§ 6.1). The monitor component is based on EEAT, a toolkit described back in Section 2.1.6, and works by compiling *AwReqs* described in OCL_{TM} into a set of rules which allow a rule-based engine to determine when an indicator has failed or succeeded (§ 6.2).

The adaptation component implements an Event-Condition-Action process that respond to evaluation of *AwReq*'s satisfiability as events, checks applicability conditions of associated adaptation strategies and executes them as actions. Each indicator can also be associated with resolution conditions, which are also considered by the component (§ 6.3). When the selected strategy is reconfiguration, the homonymous component is activated, executing the process described for the *Qualia* framework back in Section 4.2 (p. 99) and submitting the new system configuration to the target system as the adaptation instruction (§ 6.4).

Finally, the performance of the framework is also evaluated in terms of randomly-generated requirements models with scalable number of goals and parameters (§ 6.5).

Chapter 7

Empirical evaluation

Don't worry if it doesn't work right.
If everything did, you'd be out of a job.

Unknown author

In the previous chapters, we have proposed a modeling language to represent requirements for system adaptation based on a feedback loop architecture, showed how to represent these new requirements elements in GORE-based requirements specifications, introduced a systematic process to augment "vanilla"[1] specifications with these new elements and, finally, described a framework that can use such specifications to implement the generic functionalities of feedback loops, adding adaptation capabilities to target systems. At this point, we have covered most of the research questions proposed in Chapter 1 for this thesis.

In this chapter, we cover the last, remaining question, **RQ4**: *How well does the approach perform when applied to realistic settings?* As introduced in Section 1.3 (p. 12), we have applied descriptive and experimental methods from Design Science [Hevner et al., 2004]:

- *Scenarios* and *informed arguments* were used throughout the thesis, applying our proposals to a running example, the Meeting Scheduler;
- *Simulations* in a *experiment* were applied to a larger system, based on a case study adopted from the literature.

This system is called the *Adaptive Computer-aided Ambulance Dispatch System* (hereafter, A-CAD). Its analysis and design are presented in full in a technical report [Souza,

[1] As before in this thesis, by "vanilla" we mean the requirements of the system-to-be that are not related to its desired adaptation capabilities.

2012] and will be summarized in this chapter. Moreover, this chapter presents the results of simulations of failures in the A-CAD, which tested the response of the *Zanshin* framework to situations that require run-time adaptation.

7.1 The Computer-aided Ambulance Dispatch System

The failure of the London Ambulance Service Computer-Aided Despatch (LAS-CAD) system in the fall of 1992 became a well known case study in the area of Software Engineering. Following the report on the inquiry published by the South West Thames Regional Health Authority [Finkelstein, 1993], papers on the subject were published in different communications, such as the proceedings of the 8^{th} International Workshop on Software Specification and Design (IWSSD) [Finkelstein and Dowell, 1996], the European Journal of Information Systems [Beynon-Davies, 1995], the Journal of the Brazilian Computer Society [Breitman et al., 1999], ACM SIGSOFT Software Engineering Notes [Kramer and Wolf, 1996], amongst others.

Being a real system and having so much available information — due to its failure and subsequent inquiry — makes the LAS-CAD a good choice for validation of new research proposals. In effect, the focus of the discussions in the 8^{th} IWSSD was on which methods/techniques/tools should be applied in dealing with systems such as the LAS-CAD, and what research should be conducted to help in the development of such applications in the future [Kramer and Wolf, 1996]. Other examples of this use can be seen, for instance, in Letier's PhD thesis [2001] and You's masters dissertation [2004].

In particular, the LAS-CAD failure report [Finkelstein, 1993] states the following in paragraph 3024:

> *It should be said that in an ideal world it would be difficult to fault the concept of the design. It was ambitious but, if it could be achieved, there is little doubt that major efficiency gains could be made. However, its success would depend on the near 100% accuracy and reliability of the technology in its totality. Anything less could result in serious disruption to LAS operations.*

Thus, the high criticality of many of the components of the LAS-CAD make it a good case for adaptive systems, because self-adapting to failures — which invariably occur in a system that depends on near 100% reliability — is one way to avoid the aforementioned serious disruption to LAS operations. Take, for instance, the requirement of getting an ambulance to the scene of the incident as quickly as possible. In the case of the LAS, a set of standards (called ORCON) had been devised to indicate what percentage of ambulances should arrive in 3 minutes, 10 minutes and so on. There is no way to simply

put that table into the system and guarantee that the standards will be followed [Kramer and Wolf, 1996]. Instead, adaptation actions can be taken whenever the system does not satisfy such requirements.

Note, however, that is not our intention to prove that the LAS would not have failed if it had been built as an adaptive system using our proposal. Many of the analyses conducted over the failure indicate that the procurement and the development processes were flawed, producing a bad quality system in general. Hence, if adaptation mechanisms had been developed to work with the LAS, there is no guarantee these would have been properly developed and have good quality and would therefore also be prone to failure. Our objectives here are to learn from the problems detected in the LAS in order to identify critical requirements and use those to develop a new system which would, in theory, be designed properly and have good quality in general.

In the remainder of this section, we present the "vanilla" requirements for the A-CAD, based on previous publications about the LAS-CAD. As previously mentioned, a more detailed account of this requirements engineering process can be found in [Souza, 2012]. Moreover, an $i^\star$ [Yu et al., 2011] analysis of the LAS-CAD presented in the technical report "Experiences with applying the $i^\star$ framework to a real-life system", by Jane You [2001] served as early requirements analysis for the A-CAD.

7.1.1 Scope

A real CAD system is very large and complex. In our experiments, we focused on the core functions of a CAD software. We assume, therefore, that there are other systems which produce a series of events related to the ambulances managed by the CAD and which are monitored by the core CAD software to know which are available and where they are located (the dependencies between the CAD software and these other systems is shown in [You, 2001]).

Figure 7.1 shows the states an ambulance can assume during its life-cycle and the events that trigger the transitions. Below we describe these events, which are adapted from [Kramer and Wolf, 1996]. The CAD software is supposed to be aware of all such events.

- **Creation**: ambulance has been registered within the system;
- **Commissioning**: ambulance has been assigned to a station. This assignment can change over time in case of need;
- **Activation and deactivation**: ambulances can be deactivated during certain periods of time (e.g., when they need to be repaired, refueled, etc.). Deactivated ambulances cannot be dispatched;

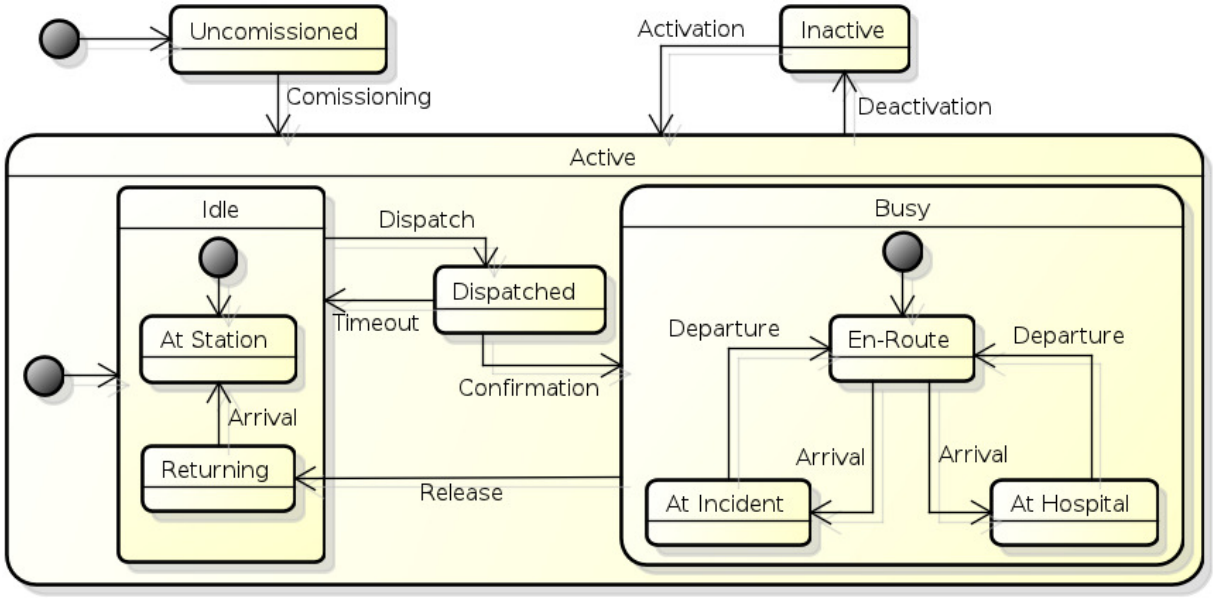


Figure 7.1: State-machine diagram for Ambulances.

- **Arrival and departure**: ambulance has arrived or has left a given location. This location can belong to a station, a hospital or an incident;
- **En-route location**: periodical reporting of location during the mobilization of ambulances to a target location. This event is monitored only for ambulances that are active and outside their stations. Each ambulance in this condition is supposed to send location updates every 13 seconds;
- **Dispatch, timeout, confirmation and release**: when an ambulance is dispatched to an incident (by the core CAD software), it should be confirmed (by its crew) so it is considered engaged to resolving the incident. This has to occur in a timely fashion, otherwise the CAD will search for another ambulance to dispatch and the first one will go back to being idle. When the incident is resolved (e.g., injured people are dropped off at the hospital) the ambulance is released and becomes idle. Only idle ambulances can be dispatched.

Furthermore, events of **commissioning** and **deactivation** should also be monitored for crew members and equipment in order to know, at any given time, what is the configuration of each ambulance. For example, a crash kit could break and be sent to repair, leaving an ambulance without it; or an EMT could take a lunch break for one hour leaving his ambulance with one less crew member for a while.

Finally, some entities and situations are considered out of the scope of the CAD system.

The following is a list of domain assumptions with respect to the requirements of the CAD software:

- **Caller information**: information about the caller and the phone used to report an emergency is added to the incident's report by the telephone operator for logging purposes only. The CAD software will not consider this information when dispatching resources and there will be no support for identifying a thread of calls from the same person;
- **Incident category**: in real CAD systems, incidents are categorized by importance. For instance, the LAS has three main categories — A (red), B (amber) and C (green) — divided in two or three subcategories each.[2] Different categories can have different standards regarding levels of service, for example. We assume, however, that all calls are of the same category;
- **Treatment**: it is not the responsibility of the CAD to follow the treatment of injured parties. In fact, the people affected by an incident are not monitored at all by the CAD, which expects only to receive a *release* event when ambulances are done with an incident. It is the responsibility of the dispatched crew to conclude when an incident is resolved and inform the CAD;
- **Dispatching to emergencies only**: ambulances only get allocated in the CAD in response to incidents. In case the service is provided by the public authorities, a separate system should manage these situations and deactivate ambulances whenever they get dispatched to non-emergencies;
- **Initial data is given**: the information required by the CAD to dispatch resources is assumed to be given: the limits of the serviced region, its division in sectors, location of hospitals and stations, list of preferred hospitals/stations for each sector, ambulances per station, ambulance crews and equipments, etc. In a real system, such information is presumably calculated and periodically modified after analyzing statistics on the amount and nature of incidents in each sector of the serviced region in the past.

7.1.2 Stakeholder Requirements

Given the above description of entities and the scope of the problem, the following is a list of requirements for the CAD software:

[2] See "Categorised", posted at the blog "Random Acts of Reality", `http://randomreality.blogware.com/blog/_archives/2004/2/18/21077.html` (last access: July 21st, 2011).

Incident Response

REQ-1. The system shall allow staff to register calls they receive from citizens;

REQ-2. The system shall, whenever possible, detect the location of the caller and associate it with the call registry (public phones have associated locations, cell phones might be triangulated, etc.);

REQ-3. The system shall allow staff to dismiss calls as non-emergencies;

REQ-4. The system shall assist staff in identifying, through the information from the call, if it refers to an open incident in the system;

REQ-5. The system shall allow staff to assign calls to open incidents as duplicates or create new incidents for calls;

REQ-6. The system shall allow staff to indicate the number of ambulances needed and their respective configurations (e.g., ambulance with paramedics, fire truck and firemen, motorcycle response unit, etc.);

REQ-7. The system shall allow staff to confirm the information related to new incidents, clearing them for dispatch by the system;

REQ-8. The system shall, upon confirmation of an incident, determine the best ambulance to be dispatched to the incident's location, given the required configuration;

REQ-9. The system shall inform stations of dispatched ambulances about the dispatching instructions, if the ambulance is in the station, or inform the ambulance itself, if it is not in the station;

REQ-10. The system shall close incidents when all resources related to it are released (see REQ-13);

REQ-11. The system shall, in case of deactivation of an ambulance that is busy, determine the best ambulance to be dispatched in replacement of the one that has been deactivated, given the required configuration. REQ-9 should follow accordingly;

REQ-12. The system shall perform in such a way that at least 75% of the ambulances arrive within 8 minutes to the location of the incident once dispatching instructions have been sent (see REQ-9). This constraint is based in the LAS-CAD standard for Category A calls.[3]

[3]See "ORCON!", posted at the blog "Random Acts of Reality", `http://randomreality.blogware.com/blog/_archives/2004/3/15/21076.html` (last access: July 21st, 2011).

Resource Monitoring

REQ-13. The system shall monitor for ambulance-related events (cf. Section 7.1.1) and keep the status of each ambulance up-to-date, including ambulance configuration;

REQ-14. The system shall show accurate and up-to-date information about on-going incidents, including status, configuration and position of engaged ambulances;

REQ-15. The system shall generate messages whenever ambulances arrive at the location of incidents, leave the location of incidents (to go to the hospital) and when they are released (incident resolved).

Exception Messages

REQ-16. The system shall generate exception messages if the dispatching process does not conclude within 3 minutes. The process is considered concluded after the number of ambulances and their configurations have been assigned (see REQ-6), the system has dispatched ambulances that fit the configuration (see REQ-8 and REQ-9) and all ambulances have confirmed the dispatch (see REQ-13);

REQ-17. The system shall generate exception messages if ambulances engaged to incidents are not released from incidents within 15 minutes of their confirmation (confirmation of the ambulance, not the incident, see REQ-13) – in other words, incidents should be resolved within 15 minutes of dispatch;

REQ-18. The system shall generate exception messages if ambulances seem to be going to the wrong direction with respect to the location they are supposed to go (see REQ-13).

7.1.3 GORE-based specification of the A-CAD

Based on the requirements elicited earlier and the $i^\star$ analysis of the LAS-CAD provided by You [2001], this sub-section finally presents the goal model that will be used as basis for the development of the A-CAD system. Figure 7.2 shows the GORE-based requirements specification (cf. Section 2.1.3, p. 32) for the A-CAD.

In the following paragraphs, we describe this model, associating its elements with the requirements that were described earlier in Section 7.1.2. The requirements IDs are shown between square brackets (e.g., [REQ-1]). Not by chance, these requirements are associated with the tasks in the model, as they represent a sequence of steps an actor (human or system) can perform to fulfill them. Moreover, all tasks in the model are associated with a requirement, showing that there are no tasks without purpose here.

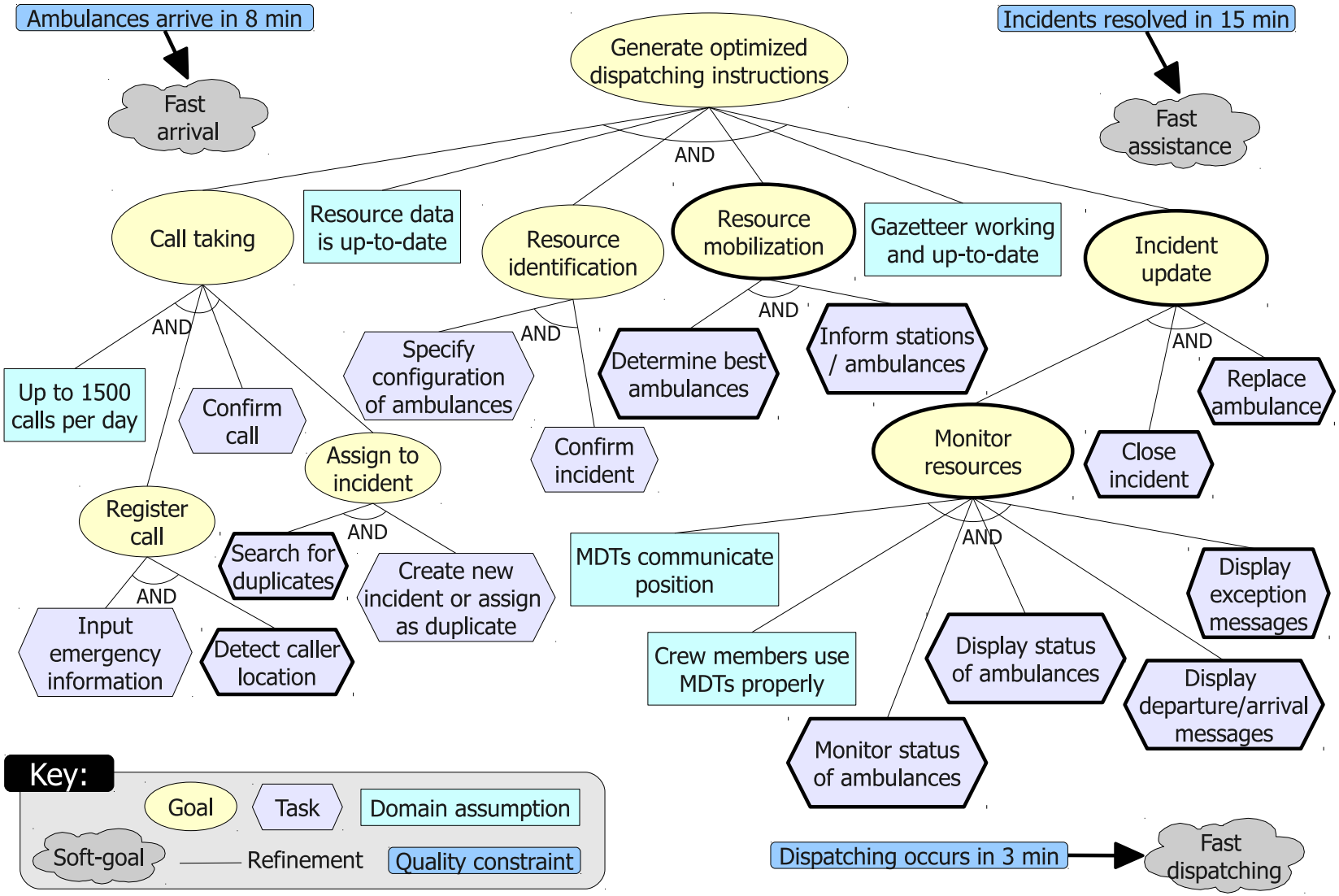


Figure 7.2: System-level goals for the CAD system-to-be.

Call taking, an activity performed mostly by staff members, consists on responding calls to the emergency service (task performed outside the system and, thus, not shown in the model), *registering* them in the CAD system, *confirming* that they are indeed emergency calls [REQ-3] and *assigning* them to an incident. During registration, the system should try to *detect the caller's location* [REQ-2] to help staff expedite the activity of *inputting emergency information* [REQ-1]. Analogously, the system should *search for duplicates* [REQ-4] to help staff decide if they should *create new incident or assign as duplicate* to an existing one [REQ-5].

Once a call has been taken and the incident registered, *resource identification* and *mobilization* are conducted for each incident. The former, performed by staff, consists on *specifying the configuration of ambulances* [REQ-6] — i.e., indicate how many ambulances should be dispatched to the incident's location and what kinds of resources (human and equipment) are needed — and *confirming the incident* [REQ-7] for dispatch by the system. *Resource mobilization* is then conducted by the system itself, *determining the best ambulance* [REQ-8] from those available and based on the provided configuration and *informing stations / ambulances* about dispatch instructions [REQ-9].

While *call taking* should be achieved for each call and *resource identification* and *mobilization* achieved for each incident, *incident update* is a goal that should be constantly maintained by the system. Categorization of goals into **achieve** and **maintain** goals have been proposed in previous works in the area of agents and multi-agent systems [Dastani et al., 2006; Morandini et al., 2009]. This means that the CAD system should attempt to satisfy this goal sub-tree periodically, at every t units of time (t to be specified during design).

To satisfy *incident update*, then, the CAD system should *monitor resources*, *close incidents* [REQ-10] when the ambulances are released and *replace ambulances* [REQ-11] that break down during service. *Monitoring resources* consists on *monitoring the status of ambulances* [REQ-13] — including all events described in section 7.1.1 — and displaying *the status of ambulances* [REQ-14], *departure/arrival messages* [REQ-15] and eventual *exception messages* [REQ-16, REQ-17, REQ-18].

For resource monitoring to work, the CAD system depends on a couple of assumptions being true. First, *MDTs [should] communicate position* of busy (engaged) ambulances at regular intervals of time (at every 13 seconds, as specified in section 7.1.1). Second, it is assumed that *crew members use MDTs properly* to notify about events in the ambulance state-chart (also see section 7.1.1) that cannot be triggered automatically by the ambulance's position, namely: commissioning, activation, deactivation, confirmation and release. Position and status of ambulances are needed in order to calculate the best ambulance to be assigned at any given time.

Finally, Figure 7.2 also shows three softgoals and their respective quality constraints that refer to time-related requirements that have been elicited from the different publications about the LAS-CAD case study:

- Dispatching, the process that starts when a call is responded and ends when ambulances acknowledge the dispatching instructions, should be done in up to 3 minutes [REQ-16];
- Once ambulances have acknowledged dispatching instructions, they should arrive at the incident's location in up to 8 minutes [REQ-12];
- The total time of assistance, which starts when an ambulance acknowledges dispatching instructions and ends when they are released from the incident, should not take more than 15 minutes [REQ-17].

It is important to note that we have modeled the system requirements so far with no variability (cf. Section 2.1.4, p. 36) whatsoever: there are no OR-refinements in Figure 7.2. This has been done on purpose to keep the model simpler at this stage and variability will be added to the requirements later during our approach.

In the next sections, we present the result of applying our systematic process for the design of adaptive systems (cf. Chapter 5) to the A-CAD. Then, at the end of the chapter, we report on the results of running simulations on the A-CAD using the *Zanshin* framework (cf. Chapter 6).

7.2 System Identification for the A-CAD

In the previous section, we have presented "vanilla" requirements for a Computer-aided Ambulance Dispatch (CAD) system. In other words, so far we have modeled the requirements of a CAD system that cannot adapt to any failures. In this chapter, we start applying our approach for the development of adaptive systems to the CAD with the objective of developing the A-CAD. We start with *System Identification*.

7.2.1 *AwReqs* for the A-CAD

In this particular experiment, we started from a requirements model for the CAD system with no variability (Figure 7.2) and proceeded to the identification of requirements and assumptions that are critical to the success of the system in order to, in a later step, attach to them certain adaptation actions that would be taken whenever the system does not satisfy such requirements (which included adding variability to the initial model).

Again using the available publications on the LAS-CAD case as source [Finkelstein, 1993; Beynon-Davies, 1995; Breitman et al., 1999; Finkelstein and Dowell, 1996; Kramer and Wolf, 1996], we analyzed what are some possible situations to which a CAD software might have to adapt in order to specify *AwReqs* to some of these situations as part of our experiment. The following list contains some CAD-related failures which were considered as possible causes for the LAS-CAD demise:

- **Misusage**: lack of cooperation from staff and crew, ranging from willful misusage to direct sabotage of the system; staff/crew members unfamiliar with the system or improperly trained to use it. This could cause crew members to use different ambulances or equipment than those specified in the dispatching instructions, crew members not pressing the appropriate buttons to confirm/release the dispatch, etc.;
- **Transmission problems**: delays or corruption of data during transmission from ambulances to the central CAD software caused by excess load on the communication infrastructure, interference with other equipment, bad coverage by the communication network in some areas (black spots), etc.;
- **Unreliable software**: errors or incorrect information produced by any of the softwares associated with the CAD system;
- **Unfamiliar territory**: dispatching of crews to parts of the serviced region they were not familiar with, which also made them drive longer to go back to the station at the end of the shift. Can cause discontentment, which triggers misusage; and longer times to resolve the incident, which could trigger exception messages;
- **Stale ambulance information**: caused by transmission problems and/or system misusage can cause the system to generate dispatching instructions which are not optimal, causing other problems such as sending crews to unfamiliar territory;
- **MDT problems**: mobile data terminals that lock up, are not readable or malfunction due to poor installation or maintenance can cause transmission problems, misusage or stale information;
- **Slow response speed**: ambulances take too long to arrive due to other problems that were already cited. This could cause citizens to call the emergency service again, increasing the number of calls. This could also cause a flood of exception messages;
- **Flood of calls**: an average amount of calls is expected everyday, but for some reason this number can significantly increase at any given day (e.g., the LAS worked

with an average of 1300-1600 emergency calls and received more than 1900 calls at the day of the failure);

- **Flood of exception messages**: exception messages should be generated when dispatching does not finish in 3 minutes [REQ-16], ambulances are not released in 15 minutes [REQ-17] or go the wrong way to the incident's location [REQ-18]. Other errors, such as transmission problems, misusage and MDT problems could cause a flood of exceptions which hinder the work of the staff.

We have thus identified 12 *AwReqs* for the A-CAD, covering most of the problems listed above. It is important to note, however, that the list of *AwReqs* is not meant to be exhaustive. The purpose of this experiment is to demonstrate that *AwReqs* can help avoiding a complete system failure by adapting to some of the situations that contributed to the LAS-CAD demise. To develop an Adaptive CAD that would be used in practice in a big city like London would most certainly require a lot more effort and elicit many other *AwReqs* in the process.

Figure 7.3 shows the goal model for the CAD previously presented in Figure 7.2 (p. 150), with added *AwReqs*. Moreover, a new task — *Get feedback*, under goal *Resource mobilization* — was also added to cope with the *unfamiliar territory* problem, as will be discussed next. Table 7.1 summarizes the elicited *AwReqs* and shows, for each of them, a short description, the CAD problem from which they originated and the pattern that represents them (cf. Section 3.1.3, p. 75).

In the following paragraphs, we justify the elicitation of each *AwReq*, explaining the rationale for its elicitation based on the CAD problems listed in the previous section.

Flood of calls: the proposed solution for the CAD represented earlier assumed that up to 1500 calls are received per day. If much more calls than that are received in any given day, something must be done so this flood of calls does not hinder the whole system, hence *AwReqs AR1* and *AR2* were elicited. The former indicates the domain assumption *Up to 1500 calls received per day* should not fail at any given day and could trigger adaptation actions to deal with a flood of calls in a particular day. The latter, by its turn, says that *AwReq AR1* should succeed 90% of the time considering month periods. This meta-*AwReq* raises awareness to the possibility that the average number of calls per day is raising and the system should evolve to normally support a bigger number of daily calls.

It is interesting to note that an aggregate *AwReq* `MaxFailure(D_MaxCalls, 0, 1d)` was used instead of a simple `NeverFail(D_MaxCalls)`. The reason for this is the following: failure of the former is registered once for the given period (1 day), whereas the latter is checked for every instance of the domain assumption verification, which would most likely

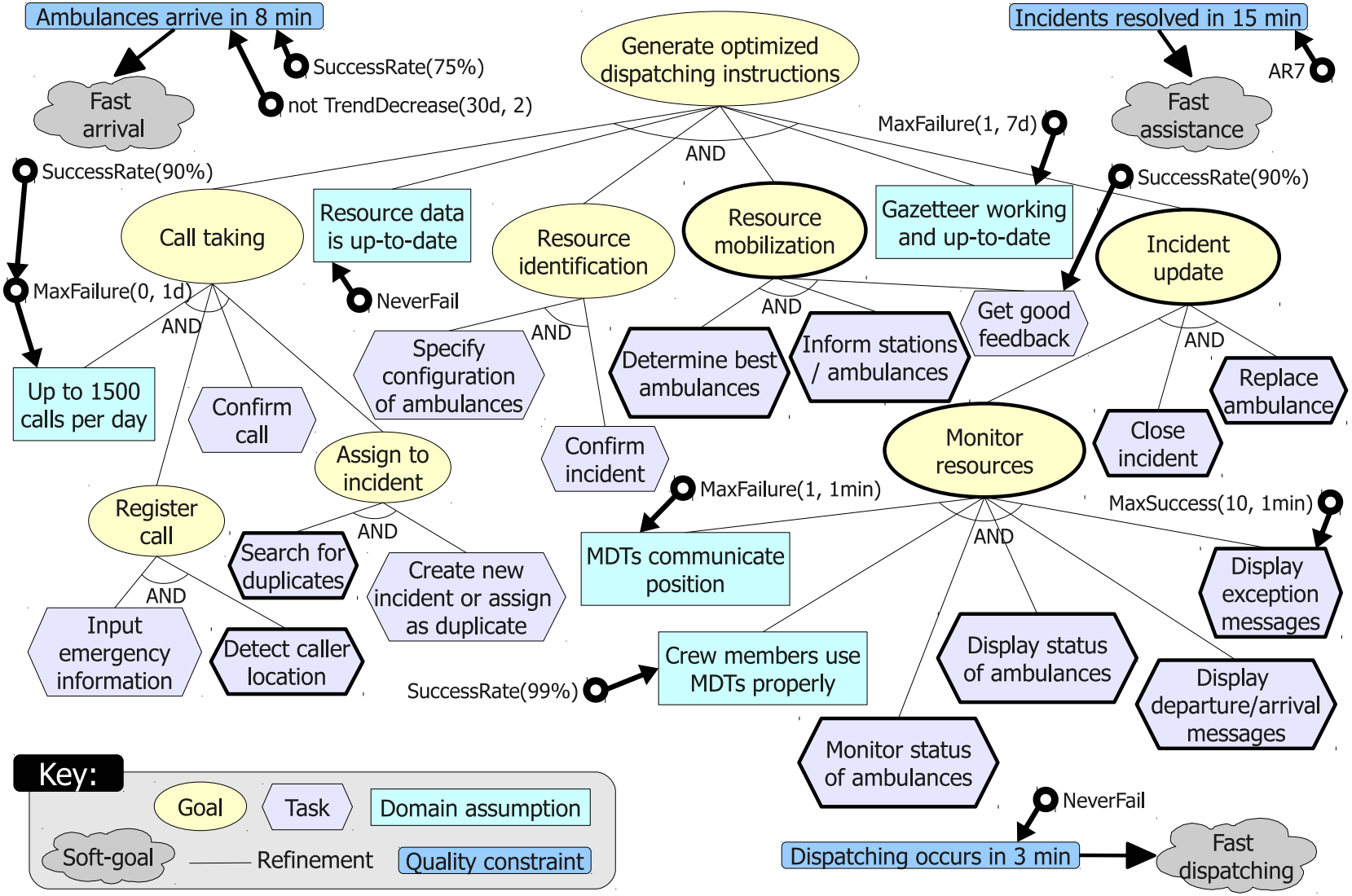


Figure 7.3: Goal model for the A-CAD system-to-be, with the elicited *AwReqs*.

Table 7.1: Summary of the *AwReqs* elicited for the A-CAD.

Id	Description	Aware of	AwReq Pattern
AR1	DA *Up to 1500 calls received per day* should not fail at any given day.	Flood of calls	MaxFailure(D_MaxCalls, 0, 1d)
AR2	AwReq AR1 should succeed 90% of the time considering month periods.	Flood of calls	SuccessRate(AR1, 90%, 1m)
AR3	QC *Ambulances arrive in 8 min* should have 75% success rate.	ORCON	SuccessRate(Q_AmbArriv, 75%)
AR4	The success rate of QC *Ambulances arrive in 8 min* should not decrease 2 months in a row.	ORCON	not TrendDecrease(Q_AmbArriv, 30d, 2)
AR5	DA *Resource data is up-to-date* should always be true.	Unreliable software	NeverFail(D_DataUpd)
AR6	DA *Gazetteer working and up-to-date* should not be false more than once per week.	Unreliable software	MaxFailure(D_GazetUpd, 1, 7d)
AR7	Task *Monitor status of ambulances* should be successfully executed with status *released* within 12 minutes of the successful execution of task *Inform stations / ambulances*, for the same incident.	Slow response	–
AR8	DA *MDTs communicate position* should not be false more than once per minute	Trans-mission	MaxFailure(D_MDTPos, 1, 1min)
AR9	DA *Crew members use MDTs properly* should be true 99% of the time.	Misusage	SuccessRate(D_MDTUse, 99%)
AR10	Task *Display exception messages* should successfully execute no more than 10 times per minute.	Flood of messages	MaxSuccess(T_Except, 10, 1min)
AR11	QC *Dispatching occurs in 3 min* should never fail.	Slow response	NeverFail(Q_Dispatch)
AR12	Task *Get good feedback* should succeed 90% of the time.	Unfamiliar territory	SuccessRate(T_Feedback,90%)

be implemented at every call. Having the meta-*AwReq* applied to the aggregate *AwReq* conveys the intended meaning of *AR2*: in 90% of the days in a month, the number of calls did not overcome the 1500 threshold. If *AR1* were not aggregate, *AR2*'s percentage would be applied to the number of calls, not the number of days!

ORCON standard: this is not one of the problems listed earlier, but a standard the LAS is supposed to follow, which we have mentioned in the beginning of section 7.1. This standard has motivated the elicitation of requirement REQ-12, which says that 75% of the ambulances should arrive within 8 minutes to the location of the incident. That is precisely what *AwReq AR3* imposes over the quality constraint *Ambulances arrive in 8 min*. Furthermore, *AwReq AR4* alerts staff about a decreasing trend in the success rate of the quality constraint, which could allow management to fix the causes of this problem before it goes lower the threshold imposed by ORCON.

Unreliable software: the CAD system depends on other software to work properly and if these are not reliable, problems are bound to arise. The standard CAD goal model thus assumes that the support system that provides data about resources and the gazetteer that provides maps of the serviced region are working properly. An *AwReq* was modeled for each of these systems: *AR5* imposes a *never fail* constraint on *Resource data is up-to-date*, whereas *AR6* tolerates one failure per week for the gazetteer.

Slow response: we divide the response of the ambulance service in two parts: dispatching, done by the staff at the central, and resolution, done by the crews in their ambulances. A constraint on the first part is depicted in the CAD model by quality constraint *Dispatching occurs in 3 min* and to indicate the criticality of this constraint, *AwReq AR11* indicates the constraint should never fail. For the second part, delta *AwReq AR7* was added to the A-CAD goal model. This *AwReq* does not have a pattern, as its definition is too specific to fit into one. It prescribes that, for each incident, the time between the ambulance or station being informed about the incident and the ambulance being released from the same incident should be no longer than 12 minutes. Counting the 3 minutes of dispatching, that gives a total of 15 minutes for incident response, as prescribed by quality constraint *Incidents resolved in 15 min*.

Transmission problems: the CAD goal model of Figure 7.3 includes the domain assumption *MDTs communicate position*, because current position of each ambulance is essential to a proper ambulance dispatch. *AwReq AR8* establishes, then, that this assumption can fail at most once per minute.

Misusage: for the CAD to work properly, it is also assumed that *Crew members use MDTs properly*. The criticality of this domain assumption is the reason for *AwReq AR9*, which prescribes a 99% success rate for it.

Flood of messages: task *Display exception messages* adds to the CAD the capability

of alerting the staff in case of different problems in the ambulance service. To cope with a possible flood of such alerts that hinders staff work, an *AwReq* was added to the amount of time this tasks succeeds in its execution. *AR10* indicates that the task should succeed at most 10 times per minute.

Unfamiliar territory: to be aware if ambulance crews are operating outside of their usual sector, a new task was added to the goal model of the CAD. *Get good feedback*, under goal *Resource mobilization*, succeeds if the crew indicates that the incident was correctly dispatched to them. Then, *AwReq AR12* establishes a 90% success rate for this task, which would alert management if more than 10% of the incidents were judged to be badly dispatched.

We can see in Table 7.1 that almost half of the *AwReqs* elicited for the A-CAD impose constraints on domain assumptions being true, which denotes the importance of adapting to changes in the environment in which the A-CAD operates. Checking if a domain assumption is true, however, may not be a trivial thing. Therefore, the following lists specifies how each of the domain assumptions should be checked:

- **Up to 1500 calls received per day:** this is the simplest assumption to be checked, as it refers to calls, which is one of the domain entities of the CAD. There are many ways of keeping the count of how many calls there have been during each 24 hour period (e.g., a query on a database of calls);

- **Resource data is up-to-date:** this assumption is deemed false if any crew or staff member reports inconsistencies between the information shown by the system and reality;

- **Gazetteer working and up-to-date:** this is checked in the same fashion as the previous assumption (data is up-to-date), plus it should be verified that the gazetteer system responds whenever it is queried;

- **MDTs communicate position:** the CAD should check that all busy (engaged) ambulances report their position at every 13 seconds;

- **Crew members use MDTs properly:** the MDT should detect and warn the CAD when things are done in violation of the proper protocol. For instance, an ambulance should not leave the station without confirmation (an incident has been assigned to it) or deactivation (for repair, etc.).

Finally, in order to avoid possible ambiguity from reading the *AwReqs*' descriptions in Table 7.1, each *AwReq* has been specified in OCL_{TM} (cf. Section 3.1.2, p. 70). Listing 7.1 shows the specification of all *AwReqs* of Table 7.1.

Listing 7.1: The *AwReqs* of the A-CAD, specified in OCL_{TM}.

```
1  package acad
2
3  -- AwReq AR1: domain assumption 'Up to 1500 calls received per day' should
       always be true.
4  context D_MaxCalls
5    inv AR1: never(self.oclInState(Failed))
6
7  -- AwReq AR2: AwReq 'AR1' should succeed 95% of the time considering month
       periods.
8  context AR1
9    def: all : Set = AR1.allInstances()
10   def: month : Set = all->select(x | new Date().difference(x.time, DAYS) <= 30)
11   def: monthSuccess : Set = month->select(x | x.oclInState(Succeeded))
12   inv AR2: always(monthSuccess->size() / month->size() >= 0.95)
13
14 -- AwReq AR3: quality constraint 'Ambulances arrive in 8 min' should have 75%
       success rate.
15 context Q_AmbArriv
16   def: all : Set = Q_AmbArriv.allInstances()
17   def: success : Set = all->select(x | x.oclInState(Succeeded))
18   inv AR3: always(success->size() / all->size() >= 0.75)
19
20 -- AwReq AR4: the success rate of quality constraint 'Ambulances arrive in 8 min
       ' should not decrease 2 months in a row.
21 context Q_AmbArriv
22   def: all : Set = Q_AmbArriv.allInstances()
23   def: m1 : Set = all->select(x | new Date().difference(x.time, MONTHS) == 1)
24   def: m2 : Set = all->select(x | new Date().difference(x.time, MONTHS) == 2)
25   def: m3 : Set = all->select(x | new Date().difference(x.time, MONTHS) == 3)
26   def: success1 : Set = m1->select(x | x.oclInState(Succeeded))
27   def: success2 : Set = m2->select(x | x.oclInState(Succeeded))
28   def: success3 : Set = m3->select(x | x.oclInState(Succeeded))
29   def: rate1 : Double = success1->size() / m1->size()
30   def: rate2 : Double = success2->size() / m2->size()
31   def: rate3 : Double = success3->size() / m3->size()
32   inv AR4: never((rate1 < rate2) and (rate2 < rate3))
33
34 -- AwReq AR5: domain assumption 'Resource data is up-to-date' should always be
       true.
35 context D_DataUpd
36   inv AR5: never(self.oclInState(Failed))
37
38 -- AwReq AR6: domain assumption 'Gazetteer working and up-to-date' should not be
        false more than once per week.
39 context D_GazetUpd
40   def: all : Set = D_GazetUpd.allInstances()
41   def: week : Set = all->select(x | new Date().difference(x.time, DAYS) <= 7)
42   def: weekFail : Set = week->select(x | x.oclInState(Failed))
43   inv AR6: always(weekFail.size() <= 1)
44
45 -- AwReq AR7: task 'Monitor status of ambulances' should be successfully
       executed with status 'released' within 12 minutes of the successful
       execution of task 'Inform stations/ambulances', for the same incident.
46 context T_MonitorStatus
47   def: related : Set = T-InformAmbs.allInstances()->select(x | x.argument("
       incident") = self.argument("incident"))
48   inv AR7: eventually(self.argument("status") = "released") and never(related->
       exists(x | x.time.difference(self.time, MINUTES) > 12))
49
50 -- AwReq AR8: domain assumption 'MDTs communicate position' should not be false
       more than once per minute.
51 context D_MDTPos
52   def: all : Set = D_MDTPos.allInstances()
53   def: minute : Set = all->select(x | new Date().difference(x.time, SECONDS) <=
       60)
54   def: minuteFail : Set = minute->select(x | x.oclInState(Failed))
55   inv AR8: always(minuteFail.size() <= 1)
56
57 -- AwReq AR9: domain assumption 'Crew members use MDTs properly' should be true
       99% of the time.
```

```
58 context D_MDTUse
59   def: all : Set = D_MDTUse.allInstances()
60   def: success : Set = all->select(x | x.oclInState(Succeeded))
61   inv AR9: always(success->size() / all->size() >= 0.99)
62
63 -- AwReq AR10: task 'Display exception messages' should successfully execute no
       more than 10 times per minute.
64 context T_Except
65   def: all : Set = T_Except.allInstances()
66   def: minute : Set = all->select(x | new Date().difference(x.time, SECONDS) <=
       60)
67   def: minuteSuccess : Set = minute->select(x | x.oclInState(Succeeded))
68   inv AR10: always(minuteSuccess.size() <= 10)
69
70 -- AwReq AR11: quality constraint 'Dispatching occurs in 3 min' should never
       fail.
71 context Q_Dispatch
72   inv AR11: never(self.oclInState(Failed))
73
74 -- AwReq AR12: task 'Get good feedback' should succeed 90% of the time.
75 context T_Feedback
76   def: all : Set = T_Feedback.allInstances()
77   def: success : Set = all->select(x | x.oclInState(Succeeded))
78   inv AR12: always(success->size() / all->size() >= 0.9)
79 endpackage
```

7.2.2 Differential relations for the A-CAD

After *AwReqs* were identified as indicators in the A-CAD, the next step in system identification was the elicitation of parameters and, then, relations between these parameters and the indicators. Differently from the Meeting Scheduler example that we have been using throughout the thesis, the A-CAD was not elicited with variability from the start and, thus, at this point in the experiment contained no variation points.

We have, thus, analyzed the *AwReqs* elicited in the previous sub-section and tried to come up with possible variability scenarios that could help in case of *AwReq* failure. During this process, the goal model has been changed to accommodate eight new parameters: control variables *NoC*, *NoSM* and *LoA* and variation points *VP1* through *VP5*, adding some new goals and tasks to the goal model as well. Figure 7.4 shows the resulting goal model from system identification, including all identified parameters. This new model also shows the name of the *AwReqs* next to their graphical representation for an easier reference in the explanations that follow.

Parameters

Control variables *NoC* — maximum *Number of Calls* that can be handled daily — and *NoSM* — *Number of Staff Members* working on the present day are associated with domain assumption *Up to 1500 calls per day*, which has also been changed and now reads *Up to* $\langle NoC \rangle$ *calls per day*, meaning the assumption is checked against the *NoC* parameter and is no longer fixed at 1500 calls.

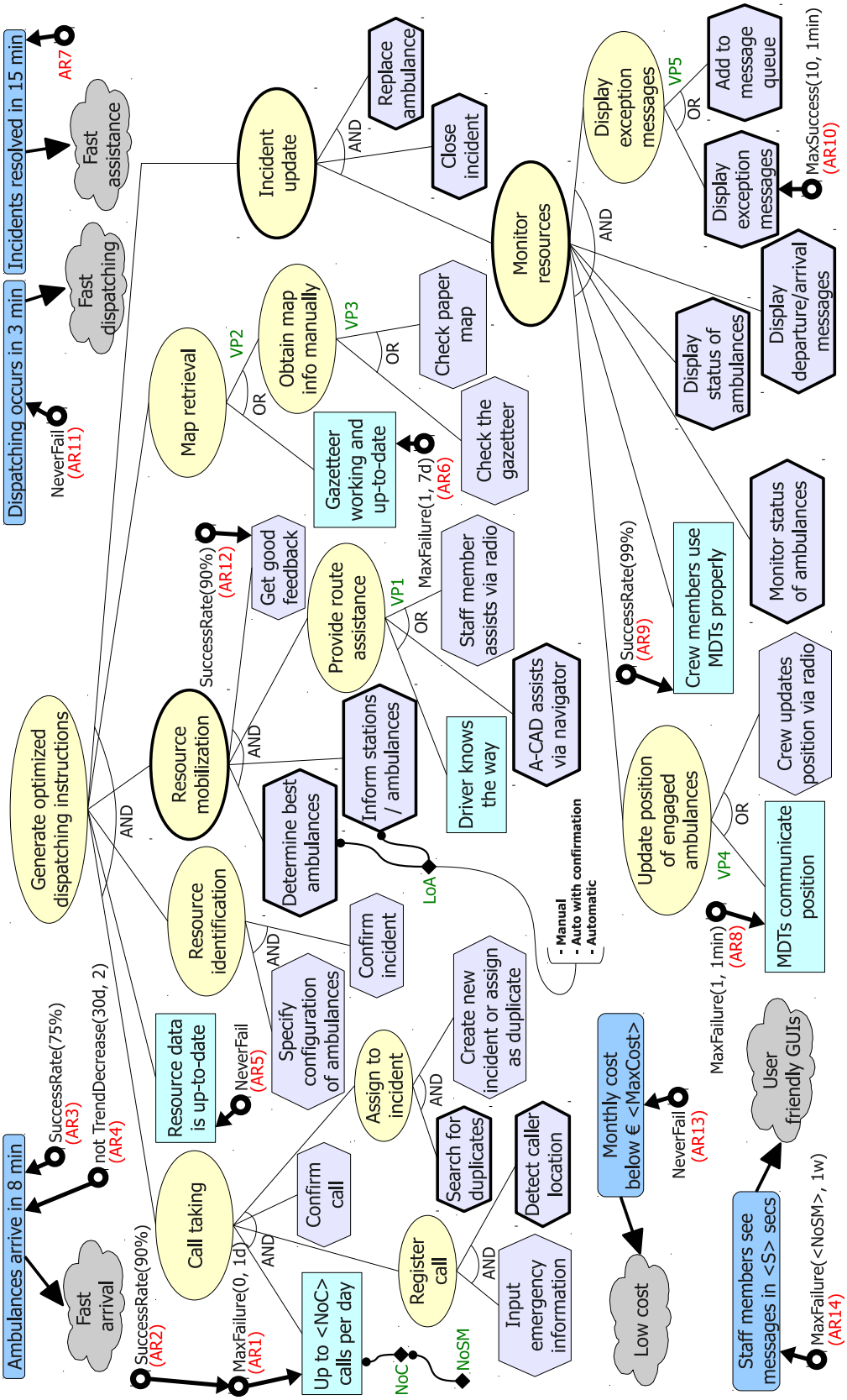


Figure 7.4: Goal model for the A-CAD system-to-be, with identified control variables and variation points.

However, parameter *NoC* is a special kind of parameter that cannot be set directly. Instead, it is declared as a direct function of another parameter, namely *NoSM*. The maximum number of calls the service can take in a day is then calculated based on the number of staff members working on that specific day. Hence, by changing the number of staff members on duty one can affect positively or negatively the success rate of the domain assumption, affecting, thus, indicators (*AwReqs*) *AR1* and *AR2*.

Control variable *LoA* — *Level of Automation* of the dispatch procedure — is an enumerated parameter that is associated with tasks *Specify configuration of ambulances* and *inform stations / ambulances* and can assume one of three values: *manual* (dispatch will be done completely manually by staff members, communicating with ambulances and stations via radio), *automatic with confirmation* (dispatch orders are suggested by the A-CAD but are sent only if a staff member confirms that is indeed the best choice) or *automatic* (the A-CAD autonomously generates dispatch orders and send them to ambulances/stations).

Changing the value of this parameter can affect indicators *AR3*, *AR4*, *AR9* and *AR12*. The rationale behind this effect is that switching to a more manual process helps solve problems that are too complicated for the A-CAD's reasoning capabilities. The interaction between the staff member in charge of the dispatch and crew members in ambulances and stations can make sure the crew agrees with the dispatching instructions (increasing the success rate of *Get good feedback*), allows for the staff member to assist the crew about the use of the MDT (increasing the success rate of *Crew members use MDTs properly*) and ultimately aid in achieving the ORCON standards (higher success rates for *Ambulances arrive in 8 min*). Obviously the benefits do not come for free: the more manual the process is, the more time each staff member spends on each incident, which allows them to take less calls a day and makes dispatching more time-consuming.

The parameter *VP1*, in its turn, was elicited to provide alternatives for improving indicator *AR7*, which talks about the time crews take to resolve an incident once the dispatching information has been received by them. One way the A-CAD can help in this matter is to provide route assistance to ambulance drivers, so they can reach the incident's location and, whenever needed, take injured people to the hospital as fast as possible. Therefore the goal *Provide route assistance* has been added to *Resource mobilization*'s AND-refinement. This new sub-goal can be satisfied in three different ways: (a) assuming that the *Driver knows the way* and, thus, doing nothing; (b) having the *A-CAD assist via navigator*; or (c) having the *Staff member assist via radio*. Again, here there is a trade-off between how personalized this assistance is and how much time it takes from staff members.

Indicator *AR7* could also be affected by changes in *LoA*. Once again, having a more

direct communication between staff member and ambulance crew can help determine the best way to reach the incident's location and resolve it.

Variation points *VP2* and *VP3* have been elicited along with a new subtree of the main goal of the system in order to include an alternative to the gazetteer for map provision, therefore affecting indicator *AR6*. The goal *Obtain map information* was added to the model where the domain assumption *Gazetteer working and up-to-date* used to be, making the assumption one of its children in OR-refinement *VP2*.

The other child — goal *Obtain map info manually* — is, in effect, the alternative to using the gazetteer automatically. When this alternative is selected, a staff member is supposed to check the map to determine the exact location of the incident and the best ambulances to be dispatched. This goal is further refined into two tasks in *VP3*: the staff member can either *Check the gazetteer* itself or, in extreme cases, *Check paper map*. Like in previous parameters, the alternatives range from highly automated to highly manual, providing a trade-off between avoiding software mistakes and the time taken by staff members for each dispatch.

Finally, there are variation points *VP4* and *VP5*. In the former, a new goal — *Update position of engaged ambulances* — has replaced domain assumption *MDTs communicate position*, making it one of its children in an OR-refinement. The other child is task *Crew updates position via radio*, which consists on a manual fall-back for when MDTs are not working properly, thus affecting indicator *AR8*. Radio contact between crew and staff also allows crew members to avoid using the MDT altogether, passing all information directly via voice. Therefore, this parameter also affects indicator *AR9*.

Parameter *VP5* provides a simple solution to the flood of exception messages, monitored by indicator *AR10*: add messages to a message queue instead of showing them directly. To this end, the task *Display exception messages* has been replaced by a homonymous goal, which is now its parent, having on the other side of the OR-refinement the task *Add to message queue*.

Differential Relations

After we identified parameters that could provide the necessary variability for the A-CAD to adapt to the previously elicited possible failures at runtime, we specified the effect that changes in these parameters have on the modeled *AwReqs* using differential relations, as prescribed in our approach. Table 7.2 shows the initial set of indicator/parameter relations for the A-CAD.

For the relations that refer to enumerated control variable *LoA* to make any sense, it is required that a total order of the parameter's enumerated values be provided. This order shall be as follows: $\langle manual \rangle \prec \langle auto\ with\ confirmation \rangle \prec \langle automatic \rangle$. Variation

Table 7.2: Initial set of differential relations of the A-CAD.

$\Delta(AR1/NoSM)[0, maxSM] > 0$	(7.1)	$\Delta(AR6/VP2) > 0$	(7.11)
$\Delta(AR2/NoSM)[0, maxSM] > 0$	(7.2)	$\Delta(AR11/VP2) < 0$	(7.12)
$\Delta(AR3/LoA) < 0$	(7.3)	$\Delta(AR12/VP2) > 0$	(7.13)
$\Delta(AR4/LoA) < 0$	(7.4)	$\Delta(AR6/VP3) > 0$	(7.14)
$\Delta(AR9/LoA) < 0$	(7.5)	$\Delta(AR11/VP3) < 0$	(7.15)
$\Delta(AR11/LoA) > 0$	(7.6)	$\Delta(AR12/VP3) > 0$	(7.16)
$\Delta(AR12/LoA) < 0$	(7.7)	$\Delta(AR8/VP4) > 0$	(7.17)
$\Delta(AR3/VP1) > 0$	(7.8)	$\Delta(AR9/VP4) > 0$	(7.18)
$\Delta(AR4/VP1) > 0$	(7.9)	$\Delta(AR11/VP4) < 0$	(7.19)
$\Delta(AR7/VP1) > 0 \Delta(AR11/VP1) < 0$	(7.10)	$\Delta(AR10/VP5) > 0$	(7.20)

points assume their default order, i.e., ascending from left to right according to their position in the model.

Most of the effects formalized by the relations were discussed in the previous step because they motivated the very elicitation of the parameters. However, at this step of the process each of the elicited parameters were again analyzed and compared to each system indicator to make sure all effects were identified and modeled. This analysis resulted in the identification of the following new relations:

- All parameters, with the exception of *VP5* and *NoSM*, have an effect on indicator *AR11*, which says that quality constraint *Dispatching occurs in 3 min* should never fail. A higher level of automation (*LoA*) improves it — Equation (7.6) —, whereas choosing to do tasks manually with the involvement of a staff member (*LoA* and *VP1* through *VP4*) has a negative effect on it — equations (7.6), (7.10), (7.12), (7.15) and (7.19);

- Variation points *VP2* and *VP3* also have an effect on indicator *AR12*, which states that task *Get good feedback* (for the dispatch choice) should succeed 90% of the time — equations (7.13) and (7.16). The rationale is that obtaining map information manually may help in the process of choosing the best ambulance to dispatch;

- Parameter *VP1*, which indicates the kind of route assistance to give ambulance drivers, also affects indicators *AR3* and *AR4*, which refer to quality constraint *Ambulances arrive in 8 min* — equations (7.8) and (7.9). Providing route assistance may help satisfy ORCON standards.

Another activity of this step is the identification of landmark values for numeric control variables, that establish intervals in which the identified relations can be applied. The only applicable numeric parameter is *NoSM* and all of its relations — equations (7.1) and (7.2) — are valid in the interval $[0, maxSM]$, $maxSM$ being a qualitative value that represents the maximum number of staff members the ambulance service infrastructure can hold. *NoC* is also numeric, but it is not applicable as cannot be directly modified (it is a function of *NoSM*). Hence, no differential relation or landmark value were identified for it.

Given the relations in Table 7.2 and assuming each of the parameters has been assigned an initial value it is possible to use the information of how parameters affect indicators at runtime to change their values whenever there is a system failure. This change, however, may require some kind of trade-off analysis at runtime. For instance, as stated before, choosing to do dispatching tasks manually (*LoA* and *VP1* through *VP4*) might improve several different indicators, but at the cost of having a negative impact over *AR11*.

Trade-offs

A careful analysis of these relations, however, will indicate that there are some indicators missing in our model of the A-CAD. After all, *AR11* is the only indicator that receives a negative impact from some of the parameter changes and this impact can be remedied by increasing *NoSM*. Therefore, why not setting everything to manual and increasing the number of staff members to the maximum? Also, if switching *VP5* to *Add to message queue* solves the flood of messages problem, why not use it exclusively?

The answer to these questions relies on some implicit quality indicators, i.e., non-functional requirements that have not been explicitly elicited. Clearly, increasing the number of staff members also increases the cost of the overall system, whereas the use of a message queue might be avoided unless strictly necessary because of user friendliness concerns. For the purposes of this experiment, we assume the existence of the following stakeholder requirements (which have already been depicted earlier in Figure 7.4):

- We should aim for *Low cost* (softgoal). In particular, stakeholders would like *Monthly cost below* €$\langle MaxCost \rangle$ (quality constraint), where $MaxCost$ is a qualitative variable representing the maximum amount of money that should be spent for the ambulance service at any given month. This requirement should never fail;
- The A-CAD should have *User friendly GUIs* (softgoal). In particular, it should be the case that *Staff members see messages in* $\langle S \rangle$ *secs*, where S is a qualitative variable representing the maximum amount of seconds between message generation and message display. For this requirement, stakeholders would like it to fail no more

Table 7.3: Differential relations for the newly elicited indicators *AR13* and *AR14*.

$$\Delta\left(AR13/NoSM\right) < 0 \tag{7.21}$$

$$\Delta\left(AR14/VP5\right) < 0 \tag{7.22}$$

Table 7.4: Refinements for the differential relations of the A-CAD

$$|\Delta\left(AR3/VP1\right)| > |\Delta\left(AR3/LoA\right)| \tag{7.23}$$

$$|\Delta\left(AR4/VP1\right)| > |\Delta\left(AR4/LoA\right)| \tag{7.24}$$

$$VP2 \neq \langle Obtain\ map\ info\ manually \rangle \rightarrow |\Delta\left(AR6/VP3\right)| = 0 \tag{7.25}$$

$$|\Delta\left(AR9/VP4\right)| > |\Delta\left(AR9/LoA\right)| \tag{7.26}$$

$$|\Delta\left(AR11/VP2\right)| > |\Delta\left(AR11/LoA\right)| > |\Delta\left(AR11/VP3\right)| > |\Delta\left(AR11/VP1\right)| > |\Delta\left(AR11/VP4\right)| \tag{7.27}$$

$$|\Delta\left(AR12/VP2\right)| \approx |\Delta\left(AR12/VP3\right)| \approx |\Delta\left(AR12/LoA\right)| \tag{7.28}$$

than $NoSM$ per week, meaning that at most there should be, in average, one failure per staff member working on the ambulance service.

Given the new indicators (*AR13* and *AR14*), new differential relations were also identified and are displayed in Table 7.5.

Relation refinement

After the initial differential relations were identified, we proceeded to the last step of system identification: relation refinement. By comparing relations associated with the same indicator, we have identified six new relations, shown in Table 7.4. When reading these comparisons, consider that the unit for *NoSM* is *one staff member* — specified $U_{NoSM} = 1$ —, so when other parameters are compared with *NoSM*, they are comparing to "hiring or laying off one staff member". Enumerated control variables and variation points (which are themselves enumerated) have a default unit of increment of choosing the next value in their given order.

Of the fourteen indicators, six had more than one parameter associated with them: *AR3*, *AR4*, *AR6*, *AR9*, *AR11* and *AR12*. All of them follow the default combination rules (homogeneous impact is additive) and no relation was added for *AR6* because *VP3* is only relevant if *VP2* is "increased" to pursue *Obtain map info manually* instead of assuming *Gazetteer working and up-to-date*.

Finally, it is important to note that the resulting model is much simplified if compared with a real ambulance dispatch system. Taking the London Ambulance System as an example, there were probably many other softgoals and quality constraints to be elicited from the stakeholders, leading to more indicators (*AwReqs*), parameters and, as a consequence, more differential relations between indicators and parameters. The A-CAD

was intentionally simplified for the purposes of this experiment (which is, after all, a laboratory demonstration and not a full-fledged case study with an industrial partner).

7.2.3 Final additions to the A-CAD model

To better illustrate some adaptation strategies in the next section, we have included a few new elements in the goal model of the A-CAD, resulting in the model shown in Figure 7.5, which is the final goal model for the A-CAD. The new elements are:

- Numeric control variable *MST* (*Minimum Search Time*), representing the minimum amount of time (in seconds) staff members must dedicate to the task of searching for duplicates;
- Softgoal *Unambiguity*, operationalized by quality constraint *No unnecessary extra ambulances dispatched*;
- *AwReq AR15*, which specifies that the goal *Register call* should never fail (`NeverFail (G_RegCall)`);
- *AwReq AR16*, imposing a comparable delta constraint that verifies that, in fact, the number of ambulances at the scene is the same number of ambulances in the configuration of the dispatch (`ComparableDelta(T_SpecConfig, Q_NoExtra, numAmb, 0)`);

MST is directly related to the new softgoal, *Unambiguity*: if staff members are forced to spend some time searching for duplicate calls, this will lower the probability of missing a duplicate and registering a call as a new incident, which would in turn result in duplicate (ambiguous) dispatch. On the other hand, the trade-off here is that higher values for *MST* may imply harming softgoals such as *Fast arrival* and *Fast dispatching*.

In its turn, *AR15* requirement represents the fact that the goal *Register call* is critical to the dispatch process, for the very simple reason that the A-CAD cannot process an incident that has not been registered into the system and, thus, the entire process will have to be conducted manually if this goal is not satisfied.

Table 7.5 shows the new differential relations and new and changed refinements added to the A-CAD model after the addition of the new elements. The following list describes the new/modified relations:

- Increasing the *Minimum Search Time* will affect negatively the success of quality constraint *Dispatching occurs in 3 min* (*AR11*) for an obvious reason: the time spent searching for duplicates could be spent with other tasks related to dispatching and incident resolution in order to finish them faster — Equation (7.30);

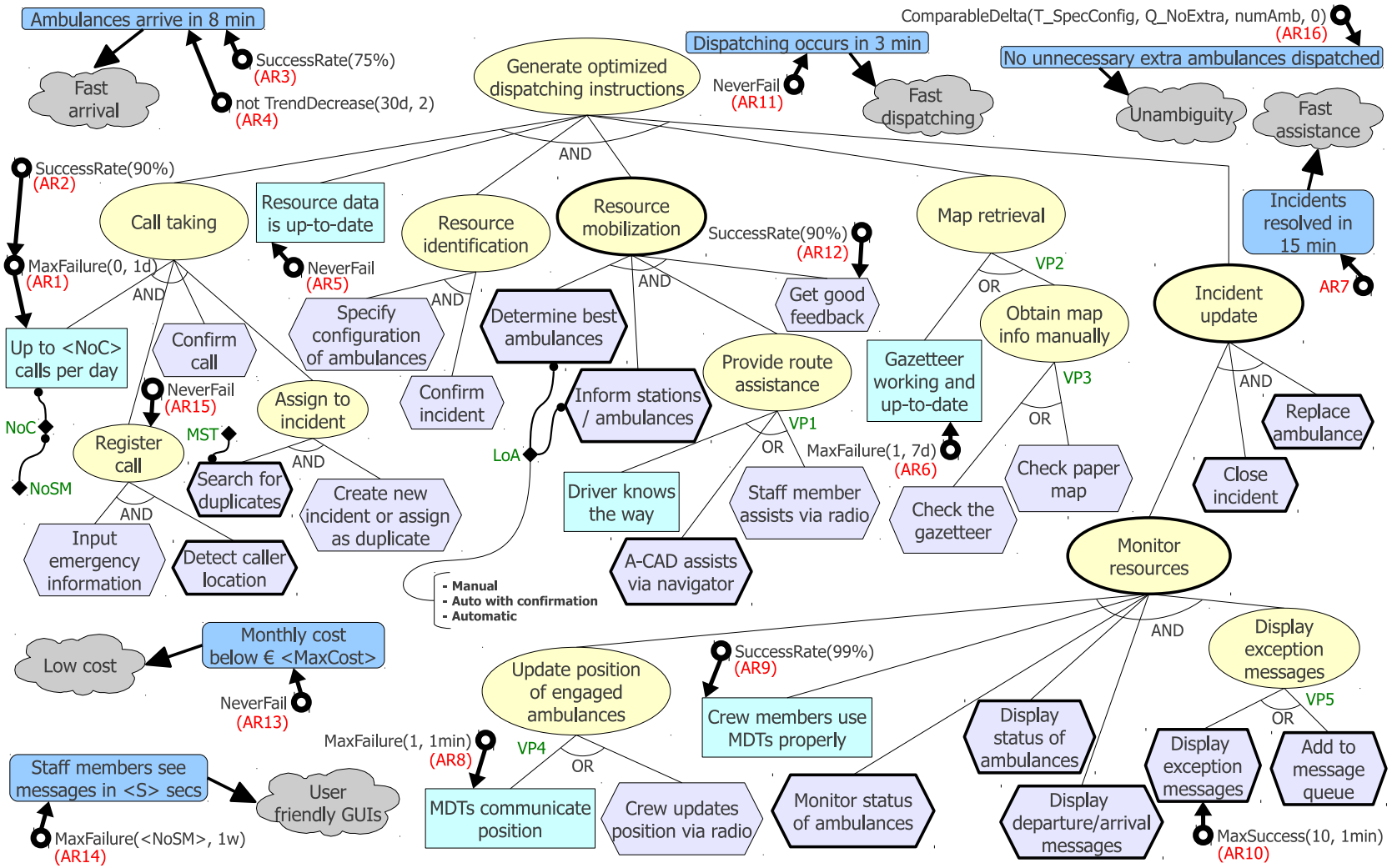


Figure 7.5: Final goal model for the A-CAD system-to-be.

Table 7.5: New differential relations and refinements, after the final additions to the specification.

$$\Delta\left(AR12/MST\right)[0,180] > 0 \tag{7.29}$$
$$\Delta\left(AR11/MST\right)[0,180] < 0 \tag{7.30}$$
$$\Delta\left(AR13/MST\right)[0,180] > 0 \tag{7.31}$$
$$\Delta\left(AR16/MST\right)[0,180] > 0 \tag{7.32}$$
$$\Delta\left(AR16/LoA\right) < 0 \tag{7.33}$$
$$\ldots > |\Delta\left(AR11/VP4\right)| > |\Delta\left(AR11/MST\right)| \tag{7.34}$$
$$|\Delta\left(AR12/VP2\right)| \approx |\Delta\left(AR12/VP3\right)| \approx |\Delta\left(AR12/LoA\right)| \approx |\Delta\left(AR12/MST\right)| \tag{7.35}$$
$$|\Delta\left(AR13/NoSM\right)| > |\Delta\left(AR13/MST\right)| \tag{7.36}$$
$$|\Delta\left(AR16/MST\right)| > |\Delta\left(AR16/LoA\right)| \tag{7.37}$$

- On the other hand, increasing *MST* affects positively *AR16* — the more time spent searching for duplicates, the less chance of an ambiguous dispatch, Equation (7.32) —, *AR12* — duplicate dispatches will most likely get bad feedback from crew members who will be sent to assist an incident unnecessarily — Equation (7.29) — and *AR13* — duplicate dispatches represent waste of resources, and therefore money, Equation (7.31);
- The *Level of Automation* also affects `AR16` (i.e., quality constraint *No unnecessary extra ambulances dispatched*): on a more manual setting staff members can check amongst themselves if the dispatch they are currently doing is ambiguous and cancel one of the dispatches before ambulances are mobilized — Equation (7.33);
- Regarding *AwReqs AR11* and *AR13*, parameter *MST* is the one with the lowest effect — equations (7.34) and (7.36). On the other hand, when dealing with *Unambiguity* (i.e., *AwReq AR16*), *MST* is better than *LoA* — Equation (7.37). For *AR12*, all parameters have roughly the same effect, including the new parameter *MST* — Equation (7.35).

7.3 Adaptation Strategy Specification for the A-CAD

Given the final goal model for the A-CAD, a complete specification of the adaptation strategies for the system can be provided. Table 7.6 presents the list of strategies associated to each *AwReq* failure.

AwReqs AR1 and *AR2* monitor if the domain assumption *Up to* $\langle NoC \rangle$ *calls per day* is true and the only way to improve the success rate of this assumption is by increasing

Table 7.6: Final specification of adaptation strategies for the A-CAD experiment.

AwReq	***AwReq* pattern**	**Adaptation strategies**
AR1	`NeverFail(T_InputInfo)`	1. *Warning("AS Management")* 2. *Reconfigure(∅)*
AR2	`SuccessRate(AR1, 90%)`	1. *Warning("AS Management")* 2. *Reconfigure(∅)*
AR3	`SuccessRate(Q_AmbArriv, 75%)`	1. *Reconfigure({Ordered Effect Parameter Choice} [order = descending])*
AR4	`not TrendDecrease(Q_AmbArriv, 30d, 2)`	1. *RelaxReplace(AR4, AR4_60Days) + StrengthenReplace(AR3, AR3_80Pct)* 2. *Reconfigure({Ordered Effect Parameter Choice} [order = ascending])*
AR5	`NeverFail(D_DataUpd)`	1. *Delegate("Staff Member")*
AR6	`MaxFailure(D_GazetUpd, 1, 7d)`	1. *Reconfigure(∅ [n = 2])*
AR7		1. *Reconfigure(∅)*
AR8	`MaxFailure(D_MDTPos, 1, 1min)`	1. *RelaxReplace(D_MDTPos_20Secs)* 2. *RelaxReplace(AR8, AR8_45Secs)* 3. *RelaxReplace(AR8_45Secs, AR8_30Secs)* 4. *Retry(60000)* 5. *Reconfigure(∅ [Immediate Resolution])*
AR9	`SuccessRate(D_MDTPos, 1, 1min)`	1. *Reconfigure({Ordered Effect Parameter Choice} [order = descending])*
AR10	`MaxSuccess(T_Except, 10, 1min)`	1. *Reconfigure(∅ [Immediate Resolution])*
AR11	`NeverFail(Q_Dispatch)`	1. *Reconfigure({Oscillation Value Calculation, Oscillation Resolution Check})* 2. *Reconfigure({Ordered Effect Parameter Choice} [order = descending])*
AR12	`SuccessRate(T_Feedback, 90%)`	1. *Reconfigure(∅)*
AR13	`NeverFail(Q_MaxCost)`	1. *Reconfigure({Ordered Effect Parameter Choice} [order = ascending, repeat policy = max 2 times])*
AR14	`MaxFailure(Q_MsgTime, <NoSM>, 1w)`	1. *Reconfigure(∅ [Immediate Resolution])*
AR15	`NeverFail(G_RegCall)`	1. *Retry(5000)* 2. *RelaxDisableChild(T_DetectCaller)*
AR16	`ComparableDelta(T_SpecConfig, Q_NoExtra, numAmb, 0)`	1. *Reconfigure({Ordered Effect Parameter Choice} [order = ascending])*

the number of staff members (*NoSM*), which in turn automatically increases the number of calls (*NoC*) the service can take per day. Since hiring and firing staff members does not seem something that should be done automatically by a software system, the first associated strategy with these *AwReqs* is to warn the ambulance service managers.

Then, reconfiguration is provided as a second strategy to try. Given that only one parameter is related to these *AwReqs* the default procedure (represented by ∅) will be used to deal with their failures. However, it is important to note that the *maturation time* of parameter *NoSM* is five days, meaning it takes that amount of time to see the results of hiring new staff (hiring and training takes time). The adaptation algorithm will wait for this amount of time before considering new failures of *AR1* or *AR2*.

AR3 and *AR4* also refer both to the same element, namely, the quality constraint *Ambulances arrive in 8 min*. To increase its success rate, the framework can choose between variation point *VP1* or control variable *LoA*, the former having a higher effect than the latter. Since *AR3* sets the threshold for the success rate of the quality constraint, it is set to use *descending order*, choosing to change first the element with greater effect. On the other hand, *AR4* just indicates a trend of decline, but the current rate could still be well over the threshold and the choice here is to use the parameters with lowest effect first, i.e., *ascending order*.

AR5 does not have any parameters associated with it and, thus, reconfiguration is not applicable. Since it refers to failures of the domain assumption *Resource data is up-to-date*, we delegate the solution to the staff member whose session of use triggered the *AwReq*, waiting for her to check the system responsible for registration of resources and fix the problem manually.

AwReq AR6 imposes a maximum failure constraint on the domain assumption *Gazetteer working and up-to-date* and the related parameters are variation points *VP2* and *VP3* and, as specified earlier in Table 7.4, *VP2* has to be changed first, otherwise changing *VP3* has no effect. However, we have specified the *number of parameters* to choose to be $N = 2$ and, thus, both parameters will be changed at the same time. This will make the A-CAD switch always from assuming proper functioning of the gazetteer to using paper maps.

The reconfiguration strategy is also applied to *AR7*, *AR8*, *AR10* and *AR14*. Because there is just one parameter that has an effect on each of these indicators, they will all use the default algorithm. With the exception of *AR7*, however, these *AwReqs* have been marked as *immediate resolution*, which means that the adaptation algorithm will consider the problem solved immediately after making the parameter change. This makes sense for *AR8* and *AR10* because changes on their associated parameters, respectively *VP4* and *VP5*, switch the system to a branch that does not contain the elements to which

the *AwReq* refers. Changing *VP5* back to *Display exception messages* also makes *AR14* irrelevant because messages would be shown immediately to staff members. The default algorithm is also the choice for *AR12*, because all of the parameters that can affect it have roughly the same effect, so one of them will be chosen randomly.

AR8, however, also has other adaptation strategies associated to it. The original specification for the transmission of ambulance positions is very strict, therefore we apply three relax strategies (one on the domain assumption itself, two on the *AwReq*) and a retry strategy to make sure it is not a temporal glitch before reconfiguring.

For *AR9*, *AR13* and *AR16* the ordered parameter choice was also selected, being used in an *ascending order* for the latter two *AwReqs*. Furthermore, for *AR13* the *repeat policy* was set to *max 2 times* so we try to reduce costs by avoiding ambiguous dispatches (increase *MST*) a few times first before firing staff members (reducing *NoSM*).

AR11 indicates that *Dispatching occurs in 3 min* should never fail. In case it does, however, two different algorithms were chosen. The first one is the *Oscillation Algorithm*, which applies only to *MST*, as it is the only numeric variable associated with *AR11*. If this algorithm is not applicable (e.g., *MST* is not incrementable), use descending order and change other related parameters.

Finally, *AR15*, like *AR5*, is also not affected by reconfiguration and is associated with two adaptation strategies: retrying goal *Register call* after 5 seconds (in case the failure is due to a temporary error in the input form) and relaxing the goal by disabling task *Detect caller location* (in case caller detection is not working), as it is not essential to ambulance dispatch (the staff member can ask the caller for her location).

7.4 Simulations of the A-CAD using the *Zanshin* framework

In the previous sections, the A-CAD's adaptation requirements were elicited using the process described in Chapter 5 and modeled using the language presented in Chapter 3, providing initial validation for our approach for the design of adaptive systems through *informed arguments* over the elicited *scenarios* of adaptation.

In this final section, we describe the last step in the validation of the approach: an *experiment*, consisting of the development and execution of *simulation* of failure scenarios of the A-CAD at runtime, using the *Zanshin* framework. The objective was to evaluate the response of the *Zanshin* framework to the simulated failures and, thus, the effectiveness of our proposals.

As explained back in Section 6.1.1 (p. 127), for *Zanshin* to be able to read the specification of the system's ("vanilla" and adaptation) requirements, they have to be represented in EMF. Listing 7.2 shows the EMF encoding of the A-CAD for two simulations that were

developed.

Listing 7.2: EMF specification of the A-CAD requirements.

```
<?xml version="1.0" encoding="UTF-8"?>
<acad:AcadGoalModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore="http://
    www.eclipse.org/emf/2002/Ecore" xmlns:acad="http://acad/1.0" xmlns:it.unitn.
    disi.zanshin.model="http://zanshin.disi.unitn.it/1.0/eca">
  <rootGoal xsi:type="acad:G_GenDispatch">
    <children xsi:type="acad:G_CallTaking">                          <!-- 0 -->
      <children xsi:type="acad:D_MaxCalls"/>
      <children xsi:type="acad:G_RegCall">
        <children xsi:type="acad:T_InputInfo"/>
        <children xsi:type="acad:T_DetectLoc"/>
      </children>
      <children xsi:type="acad:T_ConfirmCall"/>
      <children xsi:type="acad:G_AssignIncident">
        <children xsi:type="acad:T_SearchDuplic"/>
        <children xsi:type="acad:T_CreateOrAssign"/>
      </children>
    </children>
    <children xsi:type="acad:D_DataUpd"/>                            <!-- 1 -->
    <children xsi:type="acad:G_ResourceId">                          <!-- 2 -->
      <children xsi:type="acad:T_SpecConfig"/>
      <children xsi:type="acad:T_ConfIncident"/>
    </children>
    <children xsi:type="acad:G_ResourceMob">                         <!-- 3 -->
      <children xsi:type="acad:T_DetBestAmb"/>
      <children xsi:type="acad:T_InformStat"/>
      <children xsi:type="acad:G_RouteAssist" refinementType="or">
        <children xsi:type="acad:D_DriverKnows"/>
        <children xsi:type="acad:T_AcadAssists"/>
        <children xsi:type="acad:T_StaffAssists"/>
      </children>
      <children xsi:type="acad:T_Feedback"/>
    </children>
    <children xsi:type="acad:G_ObtainMap" refinementType="or">       <!-- 4 -->
      <children xsi:type="acad:D_GazetUpd"/>
      <children xsi:type="acad:G_ManualMap" refinementType="or">
        <children xsi:type="acad:T_CheckGazet"/>
        <children xsi:type="acad:T_CheckPaper"/>
      </children>
    </children>
    <children xsi:type="acad:G_IncidentUpd">                         <!-- 5 -->
      <children xsi:type="acad:G_MonitorRes">
        <children xsi:type="acad:G_UpdPosition" refinementType="or">
          <children xsi:type="acad:D_MDTPos"/>
          <children xsi:type="acad:T_RadioPos"/>
        </children>
        <children xsi:type="acad:D_MDTUse"/>
        <children xsi:type="acad:T_MonitorStatus"/>
        <children xsi:type="acad:T_DispStatus"/>
        <children xsi:type="acad:T_DispDepArriv"/>
        <children xsi:type="acad:G_DispExcept" refinementType="or">
          <children xsi:type="acad:T_Except"/>
          <children xsi:type="acad:T_ExceptQueue"/>
        </children>
      </children>
      <children xsi:type="acad:T_CloseIncident"/>
      <children xsi:type="acad:T_ReplAmb"/>
    </children>

    <!-- Softgoals. -->
    <children xsi:type="acad:S_FastArriv"/>                          <!-- 6 -->
    <children xsi:type="acad:S_FastDispatch"/>                       <!-- 7 -->
    <children xsi:type="acad:S_FastAssist"/>                         <!-- 8 -->
    <children xsi:type="acad:S_LowCost"/>                            <!-- 9 -->
    <children xsi:type="acad:S_UserFriendly"/>                       <!-- 10 -->

    <!-- Quality Constraints. -->
```

```
65      <children xsi:type="acad:Q_AmbArriv" softgoal="//@rootGoal/@children.6"/>
              <!-- 11 -->
66      <children xsi:type="acad:Q_Dispatch" softgoal="//@rootGoal/@children.7"/>
              <!-- 12 -->
67      <children xsi:type="acad:Q_IncidResolv" softgoal="//@rootGoal/@children.8"/>
            <!-- 13 -->
68      <children xsi:type="acad:Q_MaxCost" softgoal="//@rootGoal/@children.9"/>
              <!-- 14 -->
69      <children xsi:type="acad:Q_MaxTimeMsg" softgoal="//@rootGoal/@children.10"/>
            <!-- 15 -->
70
71      <!-- AwReqs. -->
72      <children xsi:type="acad:AR1" target="//@rootGoal/@children.0/@children.0"/>
            <!-- 16 -->
73      <children xsi:type="acad:AR2"/>                                    <!-- 17 -->
74      <children xsi:type="acad:AR3"/>                                    <!-- 18 -->
75      <children xsi:type="acad:AR4"/>                                    <!-- 19 -->
76      <children xsi:type="acad:AR5"/>                                    <!-- 20 -->
77      <children xsi:type="acad:AR6"/>                                    <!-- 21 -->
78      <children xsi:type="acad:AR7"/>                                    <!-- 22 -->
79      <children xsi:type="acad:AR8"/>                                    <!-- 23 -->
80      <children xsi:type="acad:AR9"/>                                    <!-- 24 -->
81      <children xsi:type="acad:AR10"/>                                   <!-- 25 -->
82      <children xsi:type="acad:AR11" target="//@rootGoal/@children.12"
            incrementCoefficient="2"> <!-- 26 -->
83        <condition xsi:type="it.unitn.disi.zanshin.
              model:ReconfigurationResolutionCondition"/>
84        <strategies xsi:type="it.unitn.disi.zanshin.model:ReconfigurationStrategy"
              algorithmId="qualia">
85          <condition xsi:type="it.unitn.disi.zanshin.
                model:ReconfigurationApplicabilityCondition"/>
86        </strategies>
87      </children>
88      <children xsi:type="acad:AR12"/>                                   <!-- 27 -->
89      <children xsi:type="acad:AR13"/>                                   <!-- 28 -->
90      <children xsi:type="acad:AR14"/>                                   <!-- 29 -->
91      <children xsi:type="acad:AR15" target="//@rootGoal/@children.0/@children.1">
            <!-- 30 -->
92        <condition xsi:type="it.unitn.disi.zanshin.model:SimpleResolutionCondition
              "/>
93        <strategies xsi:type="it.unitn.disi.zanshin.model:RetryStrategy" time="
              5000">
94          <condition xsi:type="it.unitn.disi.zanshin.
                model:MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="
                1"/>
95        </strategies>
96        <strategies xsi:type="it.unitn.disi.zanshin.
              model:RelaxDisableChildStrategy" child="//@rootGoal/@children.0/
              @children.1/@children.1">
97          <condition xsi:type="it.unitn.disi.zanshin.
                model:MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="
                1"/>
98        </strategies>
99      </children>
100   </rootGoal>
101
102   <!-- System parameters. -->
103   <configuration>
104     <parameters xsi:type="acad:CV_MST" type="ncv" unit="10" value="60" metric="
          integer"/>
105   </configuration>
106
107   <!-- Indicator / parameter differential relations. -->
108   <relations indicator="26" parameter="//@configuration/@parameters.0"
        lowerBound="0" upperBound="180" operator="ft" />
109 </acad:AcadGoalModel>
```

The entire goal tree of Figure 7.5 (p. 168) is represented in the above EMF model (lines 3–55), along with the A-CAD's softgoals (lines 58–62), quality constraints (lines 65–

69), *AwReqs* (lines 72–99). Lines 104 and 108 show, respectively, one of the parameters and differential relations of the A-CAD, used in simulation 2.

7.4.1 Simulation 1: adaptation through evolution

The first simulation involves a failure of *AwReq AR15* which, as can be seen in line 91 of Listing 7.2, refers to *Register call* as its target using EMF's syntax for references within a model, i.e., starting at the root goal, navigate to the child with index 0 (`G_CallTaking`), then in that element navigate to the child of index 1 (`G_RegCall`). The numbers in the comments next to some elements of the listing show the index of the children of the root goal, facilitating their location.

In line 92, *AR15* is specified to have a simple resolution condition — i.e., if the *AwReq* evaluation succeeded, the problem is solved — and two associated adaptation strategies, as specified earlier in Table 7.6: `Retry(5000)` (lines 93–95) and `RelaxDisableChild (T_DetectLoc)` (lines 96–98). Both strategies are applicable at most once during an adaptation session, as can be seen in their specification.

After the A-CAD specification has been represented in EMF, an implementation of the Target System Controller Service (cf. Section 6.1.1, p. 127) specifically for the A-CAD simulation has to be provided. In a real setting, this controller would be the connection between the running A-CAD and *Zanshin*, effecting the application-specific changes related to each *EvoReq* operation (cf. Table 3.5, p. 82). In our experiments, however, we have instead implemented simulations of the A-CAD system, which call the life-cycle methods expected by the monitoring infrastructure (cf. Section 6.2, p. 129) and acknowledge the reception of *EvoReq* operations, changing the requirements model as instructed.

When this simulation is ran, the A-CAD specification is read and stored in the repository and life-cycle methods referring to tasks *Input emergency information* and *Detect caller location* are sent by the simulated system. The monitoring infrastructure detects *AR15* has changed its state, and *Zanshin* conducts the ECA-based coordination process (cf. Section 6.3, p. 133), producing a log similar to the one shown in Listing 7.3. In the listing, messages are prefixed with `TS` and `AF` to indicate if they originate from the target system or the adaptation framework, respectively, which run in separate threads. This is done to resemble more closely a real life situation, in which the target system is a separate component from the adaptation framework.

Listing 7.3: *Zanshin* execution log for the *AR15* simulation.

```
1 AF: Processing state change: AR15 -> Failed
2 AF: (S1) Created new session for AR15
3 AF: (S1) The problem has not yet been solved...
4 AF: (S1) RetryStrategy is applicable.
5 AF: (S1) Selected: RetryStrategy
6 AF: (S1) Applying strategy RetryStrategy(true; 5000)
```

```
 7 TS: Received: new-instance(G_RegCall)
 8 TS: Received: copy-data(iG_RegCall, iG_RegCall)
 9 TS: Received: terminate(iG_RegCall)
10 TS: Received: rollback(iG_RegCall)
11 TS: Received: wait(5000)
12 TS: Received: initiate(iG_RegCall)
13 AF: (S1) The problem has not yet been solved...
14
15 AF: Processing state change: AR15 -> Failed
16 AF: (S1) Retrieved existing session for AR15
17 AF: (S1) The problem has not yet been solved...
18 AF: (S1) RetryStrategy is not applicable
19 AF: (S1) RelaxDisableChildStrategy is applicable.
20 AF: (S1) Selected: RelaxDisableChildStrategy
21 AF: (S1) Applying strategy RelaxDisableChildStrategy(G_RegCall; Instance level
       only; T_DetectLoc)
22 TS: Received: suspend(iG_RegCall)
23 TS: Received: terminate(iT_DetectLoc)
24 TS: Received: rollback(iT_DetectLoc)
25 TS: Received: resume(iG_RegCall)
26 AF: (S1) The problem has not yet been solved...
27
28 AF: Processing state change: AR15 -> Succeeded
29 AF: (S1) Retrieved existing session for AR15
30 AF: (S1) The problem has been solved. Terminate S1.
```

The log shows the adaptation framework receiving notification of *AR15*'s failure (line 1), creating a new adaptation session `S1` for it (2) and searching for a suitable adaptation strategy to be applied, executing the `Retry(5000)` strategy (4–6). Then the simulated target system acknowledges the reception of the commands included in that pattern's definition (7–12), and the adaptation framework verifies that the problem has not yet been solved (13).

After a while, the monitoring component notifies one more failure of *AR15* (line 15), prompting the adaptation framework to retrieve the same adaptation session `S1` as before, realizing that it has not yet been solved (16–17). *Zanshin* then proceeds to searching for a suitable adaptation strategy, but `Retry(5000)` cannot be used again in the same session due to its applicability condition (18). The framework ends up selecting `RelaxDisableChild(T_DetectCaller)` and executing it (19–21), which again is recognized by the target system controller (22–26).

Finally, the monitoring infrastructure indicates that *AR15* has been satisfied (line 28), so the adaptation process can retrieve session `S1`, mark the problem as solved and terminate it. From this point on, further failures of *AR15* from the same user will create a new adaptation session.

As this simulation demonstrates, the framework is able to execute the specified adaptation strategies, sending *EvoReq* operations to the target system, which should then adapt according to the instructions.

7.4.2 Simulation 2: adaptation through reconfiguration

The second simulation involves the failure of *AwReq AR11* which, as specified in its `target` attribute (line 82 of Listing 7.2), refers to quality constraint *Dispatching occurs in 3 min* (look for the `<!-- 12 -->` comment to locate the 12$^{\text{th}}$ child of the root goal, line 66).

AR11 uses *Qualia* as reconfiguration strategy (line 84), with default algorithm. As explained back in Section 6.4 (p. 137), the reconfiguration strategy has to be associated with special resolution and applicability conditions (respectively, lines 83 and 85). Moreover, *AR11* also defines its increment coefficient $K_{AR11} = 2$.

In lines 103–105, the initial system configuration specifies the existing parameters and their values. Line 104 defines numeric control variable (`ncv`) *MST*, with unit of increment $U_{MST} = 10$, initial value `60` and `integer` metric, which tells *Qualia* how to perform increments. Finally, the specification includes a differential relation between *AwReq AR11* and *MST*, with lower bound set to `0`, upper bound set to `180` and `ft` (fewer than) as operator, i.e., $\Delta\,(AR11/MST)\,[0, 180] < 0$ (line 108).

When ran, the simulation produces a log similar to the one shown in figure 7.4. Here, `S` represents the target system (simulation), `Z` refers to *Zanshin* and `Q` is for *Qualia*.

Listing 7.4: *Zanshin* execution log for the *AR11* simulation.

```
1  S: A dispatch took more than 3 minutes!
2  Z: State change: AR11 (ref. Q_Dispatch) -> failed
3  Z: (S1) Created new session for AR11
4  Z: (S1) Selected strategy: ReconfigurationStrategy
5  Z: (S1) Exec. ReconfigurationStrategy(qualia; class)
6  Q: Parameters chosen: [CV_MST]
7  Q: To inc/decrement in the chosen parameters: [20]
8  S: Instruction received: apply-config()
9  S: Parameter CV_MST should be set to 40
10 Z: (S1) The problem has not yet been solved...
11
12 S: A dispatch took more than 3 minutes!
13 Z: State change: AR11 (ref. Q_Dispatch) -> failed
14 Z: (S1) ...
15 Q: Parameters chosen: [CV_MST]
16 Q: To inc/decrement in the chosen parameters: [20]
17 S: Instruction received: apply-config()
18 S: Parameter CV_MST should be set to 20
19
20 S: A dispatch took less than 3 minutes.
21 Z: State change: AR11 (ref. Q_Dispatch) -> succeeded
22 Z: (S1) Problem solved. Session will be terminated.
```

It can be seen from the figure that when *Zanshin* is made aware of the failure in *AR11* (line 2), it executes the strategy associated to this indicator in the specification (line 5), delegating the adaptation to *Qualia*. The latter, in its turn, chooses randomly the parameter *MST* (line 6), decreasing it by $V = K_{AR11} \times U_{MST} = 20$ two consecutive times (to `40` in lines 7–9, and then to `20` in lines 16–18), until the problem is deemed solved by the simulation (lines 21-22).

As this simulation demonstrates, *Zanshin* and *Qualia* are able to determine a new configuration for the target system using the information encoded in the requirements specification, instructing the system on how to reconfigure itself in order to adapt.

7.5 Chapter summary

In this chapter, we described the steps taken during an empirical evaluation of the *Zanshin* approach for the design of adaptive systems presented throughout this thesis. The evaluation consisted in modeling the adaptation requirements for an Adaptive Computer-aided Ambulance Dispatch (A-CAD) system based on a well-known case study from the literature (the LAS-CAD) and simulating run-time failures of this system to validate that our prototype framework responds accordingly.

First, we provide an overview of the problem of ambulance dispatch (§ 7.1), establishing its scope (§ 7.1.1), basic requirements in terms of "SHALL" statements (§ 7.1.2) and finally producing a GORE-based specification in terms of a goal model, as done before with the Meeting Scheduler (§ 7.1.3).

Next, we conduct our proposed *System Identification* process based on the system's "vanilla" requirements (§ 7.2), identifying *AwReqs* as indicators based on the problems that were related with the LAS-CAD (§ 7.2.1) and then modeling parameters of the system and their effect on the identified indicators using our proposed qualitative language based on differential relations (§ 7.2.2).

After System Identification, the *Adaptation Strategy Selection* activity was conducted for the A-CAD associating to each *AwReq* a list of adaptation strategies to be executed at runtime once failure of these indicators are detected (§ 7.3). Finally, simulations of failures of the A-CAD were created and executed along with our proposed framework (§ 7.4) showing that both evolution (§ 7.4.1) and reconfiguration (§ 7.4.2) are possible using our approach.

7.5.1 Evaluation conclusions

From the results summarized above, we have concluded that the *Zanshin* approach can be applied for the design of adaptive systems and that the *Zanshin* framework can be used to operationalize adaptation in target systems at runtime.

However, these conclusions are made under many assumptions, representing threats to the validity of our evaluation, the most important of which are:

- We assume that practitioners other than the author can be trained and successfully apply the *Zanshin* approach to design adaptive systems. Further development of

the approach itself (CASE tools, patterns, etc.) and surveys with practitioners are necessary to evaluate this assumption;

- We assume that target systems can be implemented or modified in order to provide our framework with the required logging information for monitoring and to perform the application-specific adaptation actions when instructions are received at runtime. Experiments with running systems instead of simulations are necessary to evaluate this assumption.

Later, in Section 8.2, we discuss in more depth the limitations of the proposals presented in this thesis.

Chapter 8

Conclusions and future work

Simplicity does not precede complexity, but follows it.

Alan Perlis

This thesis presented a requirements-based approach for the design of adaptive systems centered on the concept of feedback loops. Throughout its chapters, we have presented new modeling elements to represent requirements for adaptation (*Awareness Requirements*, *differential relations*, *Evolution Requirements*, etc.), a systematic process for requirements engineering of adaptive systems (*System Identification*, *Adaptation Strategy Specification*) and architectural considerations on how to use the produced requirements models at runtime to operationalize the feedback loop that provides adaptation capabilities to the target system. Moreover, we have applied our approach in an experiment based on a well-known case study in the Software Engineering literature.

In this chapter, we conclude the thesis summarizing what we consider to be its contributions and limitations and listing ideas for future work, some of which are already underway in our research group.

8.1 Contributions to the state-of-the-art

The contributions of this thesis can be grasped from the research questions proposed in Chapter 1 (see also Section 1.4.4 — Contributions — in page 20):

RQ1: What are the requirements that lead to the adaptation capabilities of a software system's feedback loop?

RQ2: How can we represent such requirements along with the system's "vanilla" requirements?

RQ3: How can we help software engineers and developers implement this requirements-based feedback loop?

RQ4: How well does the approach perform when applied to realistic settings?

As mentioned above, this thesis answers these questions by proposing new modeling concepts inspired by feedback control loops, a process for modeling of requirements for adaptation using these new concepts, a framework that implements the generic functionality of the feedback loop based on requirements models and preliminary validation results of the entire proposal (models and run-time framework) through simulated experiments.

It is important, however, to compare our proposals with other approaches that have been published in the literature, such as the ones cited back in Section 2.2 (p. 46), showing in more detail how our approach contributes to the state-of-the-art on adaptive systems design and development. This is done in the following paragraphs.

Compared to architecture-based approaches for adaptive systems (Section 2.2.2, p. 48), including the proposals on Autonomic Computing (Section 2.2.1, p. 47), our work differs from those by focusing on the requirements for adaptation. In other words, we propose that adaptation capabilities be considered early in the software development process, during Requirements Engineering.

Some of these approaches do provide the means to represent system requirements: Sykes et al. [2007, 2008] use finite state machines, the SASSY Framework [Menasce et al., 2011] uses a language based on BPMN, Hebig et al. [2010] use UML diagrams, etc. Our approach, however, is based on Goal-Oriented Requirements Engineering (GORE), the advantages of which were discussed back in Section 2.1.1 (p. 25).

On the other hand, our only contribution towards the architecture of the adaptive system is the assumption that adaptation will be operationalized by a feedback control loop, whose generic functionalities were implemented in the framework presented in Chapter 6. In this sense, architecture-based approaches and ours can be complimentary, provided the necessary transitions between processes and translations between models. In fact, in our research group we have already started investigating this possibility considering the Rainbow framework [Garlan et al., 2004].

As shown back in Section 2.2.3 (p. 50), however, there are many proposals for the design of adaptive systems that, like ours, focus on Requirements Engineering and, in particular, GORE. However, as also noted by Brun et al. [2009], most of these proposals do not make the feedback loop that implements the adaptation explicit. We consider feedback loops to be, in one form or another, at the core of the adaptation mechanism and, thus, our proposal differs from current works by making feedback loops first class citizen in a requirements language for self-adaptive systems.

In practice, this represents a fundamental difference between the approaches in the literature and our proposal. In the former, by default, requirements are treated as invariants that must always be achieved. Some approaches — e.g., [Baresi et al., 2010; Whittle et al., 2010] — allow you to relax non-critical goals, i.e., those that can be violated from time to time. Then, the aim of those methods is to provide the machinery to conclude at runtime that while the system may have failed to fully achieve its relaxed goals, this is acceptable. So, while relaxed goals are monitored at runtime, invariant ones are analyzed at design time and must be guaranteed to always be achievable at runtime.

In our approach, on the other hand, we accept the fact that a system may fail in achieving any of its initial (level 0) requirements. We then suggest that critical requirements are supplemented by *Awareness Requirements* (*AwReqs*) that ultimately lead to the introduction of feedback loop functionality into the system to execute compensation/reconciliation actions (in the form of *Evolution Requirements*, a.k.a. *EvoReqs*) when their failure is detected.

Another contribution of our work is related to the usability of the approach. It is known that more formal specifications yield more powerful reasoning schemes at the price of higher specification effort and lower usability by non-experts [van Lamsweerde, 2001]. We have, thus, proposed modeling concepts and chosen a specification language that are only "formal enough" to represent the requirements for a feedback loop-based adaptation and, thus, do not become a burden for requirements engineers and other developers.

Compared with approaches that advocate the use of Linear Temporal Logic (e.g., [Zhang and Cheng, 2006; Nakagawa et al., 2011; Heaven and Letier, 2011]), Fuzzy Branching Temporal Logic (e.g., [Baresi et al., 2010; Whittle et al., 2010]) or Discrete Time Markov Chains [Filieri et al., 2011], our approach is less heavy-handed in the formalism that is used, improving, thus, its usability by the average developer.

Taking, for instance, *RELAX and LoREM* (p. 51) or *FLAGS* (p. 52), it results that our approach is much simpler in comparison. The *AwReqs* constructs that we provide just reference other requirements and, thus, we believe that it is more suitable, for instance, for requirements elicitation activities. Also, our specifications do not rely on fuzzy logic and do not require a complete requirements specification to be available prior to the introduction of *AwReqs*, parameters, differential equations and *EvoReqs*.

Moreover, the language for specifying *AwReqs* does not require complex temporal constructs, even though its underlying formalism (OCL_{TM} and EEAT) provides temporal operators, so temporal properties can be expressed and monitored. In their turns, our Δ equations are abstractions based on simple concepts from Calculus and *EvoReqs* are specified using a small set of operations that are reflected into application-specific behavior in the target system. Finally, most of the work on generating specifications can be

simplified, and could even be automated, through the use of patterns.

Another interesting characteristic of our proposal, which also contributes to its usability and applicability, is that it is based on ideas from Qualitative Reasoning, as briefly discussed in Section 2.1.7 (p. 45). After all, given the usual high levels of uncertainty of problem domains (cf. Section 1.1, p. 1), which lead to incomplete knowledge about the behavior of the system-to-be, quantitative estimates at requirements time are usually unreliable [Elahi and Yu, 2011].

Therefore, we propose that qualitative information be used instead. Our approach allows the modeler to start with minimum information available and add more as further details about the system become available, either by elicitation or through run-time analysis once the system is executing. The *Qualia* framework (cf. Section 4.2, p. 99), for instance, provides a family of algorithms that require different levels of precision in terms of the relations between system parameters and indicators.

As described in Chapter 1, our work proposes two means of adaptation: *reconfiguration* and *evolution*. The latter consists of *EvoReqs* that prescribe changes on the requirements models at runtime, which are propagated to the target system through the proper means of communication. Proposals such as FLAGS [Baresi et al., 2010] or the work of Fu et al. [2010] also provide commands such as *retry*, *add/remove/modify a goal*, *relax*, etc. Our approach, however, is more flexible in the sense that *EvoReqs* are specified using a set of operations that can be used to compose different strategies and could be extended, provided support to the new operations is added to the target system.

In its turn, *reconfiguration* relates to many approaches in the literature, as shown in section *Reconfiguration approaches*, on page 55. As discussed in Chapter 6, *Qualia* was built as a component, offering a *Reconfiguration Service* implementation based on the interface defined by the *Zanshin* framework. Other reconfiguration approaches could also be plugged into *Zanshin*, provided that the requirements models contain the information required by them.

As we can see, another interesting feature of our approach is being extensible: new *AwReq* patterns can be written in OCL_{TM}, new *EvoReq* patterns can be specified using the set of operations, which are also extensible, new *Qualia* procedures and algorithms can be proposed, new components can be plugged into the *Zanshin* framework, providing new implementation to services, etc.

Furthermore, our approach is generic and can be applied to any kind of system, as opposed to other approaches in the literature, that focus on particular types of applications, such as, for instance, service-oriented applications [Pasquale, 2010; Qureshi and Perini, 2010; Peng et al., 2010], agent-oriented architectures [Morandini, 2011], etc. (this, of course, leads to one of the main limitations of our approach, discussed in the next section,

related to the degree of responsibility left in the hands of the developers of the target system). Finally, our proposals are not based on any particular RE methodology, but on concepts from the ontology of Jureta et al. [2008], which can be mapped to different GORE methods such as KAOS, $i^\star$ or Tropos.

In summary, by combining GORE, concepts from Feedback Control Theory and Qualitative Reasoning into a comprehensive approach for the design of adaptive systems, we believe this thesis provides a sound contribution towards the state-of-the-art in the field. Our work, however, is not without limitations and can be further improved or built upon in the future. We discuss these issues in the following sections.

8.2 Limitations of the approach

While our approach provides modeling elements, a systematic process and a framework which can aid developers in designing and implementing adaptive systems, this assistance is also limited in several aspects. The following list provides a summary of limitations that we have identified for the approach:

- **Target systems**: earlier we have mentioned that our approach is generic and can be applied to any kind of system. The down side of this feature is that developers are responsible for implementing all the application-specific logic, including logging (for monitoring) and the effect of *EvoReq* operations (for adaptation). On the other hand, approaches that are specific for some kinds of architectures can harness what the architecture has to offer. For instance, service-oriented approaches can use existing tools for service lookup, composition and orchestration.

  Consider, in particular, the case of socio-technical systems, which have high participation of human and organizational actors in the satisfaction of the system's requirements (cf. Section 1.2.1, p. 5). Whereas software-based functionalities can be instrumented to provide *Zanshin* with the required monitoring information, human-based features of the system require some kind of sensor to be installed to inform the framework when human-performed tasks have been initiated and completed, satisfiably or not. An analogous problem poses itself for the adaptation part of the feedback loop, if a human is to perform the adaptation action. Our framework, although applicable also in these situations given this required infrastructure, does not provide any help in designing it, which places a certain amount of burden on architectural designers and implementors.

- **Centralization**: an analogous issue comes from the fact that the *Zanshin* framework centralizes the control of the feedback loop, whereas some systems (again, this is

often the case in socio-technical systems) are composed of independent self-organized agents which collaborate towards a resulting adaptive system. Self-organization is the focus of conferences such as the International Conference on Self-Adaptive and Self-Organizing Systems (see [Brueckner and Geihs, 2011] for its most recent edition). Our feedback-loop based approach could be used to design independent software-based agents who would adapt themselves in response to failures, but we do not provide any insight on their interrelation if a centralized feedback loop cannot be enforced. Further investigation is required to analyze the relation of our work with proposals on self-organized adaptive systems.

- **Domain models**: existing approaches, such as FLAGS [Baresi et al., 2010], CARE [Qureshi and Perini, 2010] or Tropos4AS [Morandini et al., 2009], propose that the entities of the problem domain be represented in domain models that can be referred to by requirements. So far, we have not proposed any particular way of doing this, therefore analysts are responsible for this integration.

  Take, for instance, *AwReq AR11* from the Meeting Scheduler example (cf. Table 3.4, p. 77), which refers to the time the meeting has been scheduled to occur. Its specification (cf. Listing 3.1, p. 73) refers to an argument `meeting` from class `Meeting` with an attribute `startTime`, but the approach dos not provide any particular way of providing this argument to the method call monitored by EEAT.

- **Consistency and correctness**: moreover, our approach does not provide any process of technique to help analysts guarantee the consistency and correctness of the requirements models, leaving this responsibility at their hands. The use of ECA rules in *EvoReqs* constitutes a significant limitation, as large rule sets are hard to evolve, as it becomes increasingly difficult to understand what does a change entail. Moreover, attention needs to be paid to the case where conflicting rules fire at the same time.

  Requirements engineers should, in any case, guarantee that the produced specification correctly represents stakeholder requirements, but tools to assist in this task could be developed, such as, for instance, a knowledge base that can answer queries about the requirements, as in [Ernst et al., 2011].

- **Modeling notation**: as mentioned in Chapter 3, the visual notations proposed for modeling the new concepts included in our approach have been created using simple analogies with concepts such as an observing eye, a diamond operator in a programming language and the red cross. However, a more systematic methodology for the construction of visual notations (e.g., [Moody, 2009]) could be applied in

order to review the quality of the proposed notation.

- **CASE tool**: to help developers build the augmented GORE-based specifications required by the *Zanshin* framework, a CASE tool could be provided. We have started an Eclipse-based tool for this purpose, called Unagi,[1] which at the time of the publication of this thesis is still at very early stages of development.

- **Framework prototype**: the *Zanshin* framework (including *Qualia* as reconfiguration service) has been developed with the purpose of experimenting with requirements specifications produced by our approach for the design of adaptive systems at runtime, in order to verify if, based on the requirements models, a feedback loop could provide sensible results in terms of adaptation actions in response to run-time failures. Its implementations is, however, a prototype of a full feedback loop framework that can be applied to systems in real settings. Further development is needed in order to consider its use in industrial settings.

- **Legacy systems**: our approach requires a GORE-based specification of the system requirements, plus a way to monitor the system's log to detect *AwReq* failures and send it *EvoReq* operations to adapt it to these failures. These prerequisites might be difficult to attain in the case of legacy systems and techniques for assisting the developers in this case (e.g., reverse engineering as used in [Wang and Mylopoulos, 2009]) have not been studied.

- **Experiments**: this thesis reports on experiments with exemplars based on simulations of specific scenarios of run-time adaptation to evaluate if the proposed models and framework can, indeed, provide adaptivity to a target system. However, many other kinds of experiments are needed in order to provide a more complete validation of the approach.

  Surveys with developers can evaluate the proposed systematic approach and modeling language, whereas the use of real applications instead of simulations would make for a stronger case for the framework's effectiveness. Moreover, full-fledged case studies with industrial partners would be advised before taking the results of this research to industrial settings.

- **Implementation of goal models**: it has already been mentioned that our approach concentrates on Requirements Engineering and not on architectural design

[1] The name is a reference to episode 17 of the 7$^{\text{th}}$ season of American sitcom *Friends*, in which the character Ross Geller confuses the word's meaning (freshwater eels) with that of *Zanshin* (a state of total awareness). See `https://github.com/vitorsouza/Unagi`.

or implementation. Nonetheless, some assistance towards mapping goal model elements (tasks) to implemented components could help developers read GORE-based specifications when providing logging information that is used by the framework at runtime.

Consider, for example, the final specification for the A-CAD system-to-be in Figure 7.5 (p. 168). For this system, most tasks can be associated with specific components in the implemented CAD system and their instances would correspond to every dispatch that has to be done using the system. However, for any given dispatch, tasks under the goal *Monitor resources* may be executed several times, whereas a task such as *Replace ambulance* may be needed only under certain conditions (if the dispatched ambulance breaks). Currently, this mapping is also under the responsibility of developers.

- **Independence of variables**: as presented in Chapter 4, we propose a language that relates changes in single parameters to single indicators of requirements convergence in a qualitative way using differential relations. Such relations can, later, be combined to compare the effect of different parameters towards the same indicator or to establish if they could be combined for an increased effect. However, this representation greatly simplifies the actual behavior of complex, adaptive systems, whose variables (be them parameters or indicators) cannot be assumed to be independent of one another.

  Nonetheless, this simplification is not accidental. State-of-the-art methods for modeling and controlling MIMO systems — such as state/output feedback and Linear Quadratic Regulator (see [Zhu et al., 2009], § 3.4) — can be very complex and many software projects may not dispose of the necessary (human/time) resources to produce models with such degree of formality. As mentioned in the previous section, our approach is intended to be less heavy-handed in the formalism, while at the same time allowing analysts to model the requirements for the system's adaptation based on a feedback loop architecture.

8.3 Future work

The previous section highlighted several limitations of our proposal, all of which could be considered an opportunity for future work. Moreover, some questions have presented themselves along the development of this research, some of which are: what is the role of contextual information in this approach? How could we add predictive capabilities or probabilistic reasoning in order to avoid failures instead of adapting to them? Could this

approach help achieve software evolution (in the sense of software maintenance)? These and other questions show how much work there is still to be done in this research area.

We are currently investigating two directions in particular: the case of multiple concurrent indicator (*AwReq*) failures and, as mentioned earlier, the adoption of ideas from architecture-based adaptation frameworks, such as the Rainbow project [Garlan et al., 2004], so that proposed adaptations take into account what is a feasible change at the architectural level and what is not. In the following paragraphs, we sketch some early ideas on the former.

8.3.1 Considering concurrent indicators (*AwReqs*) failures

The single indicator scenario implemented by *Qualia* (cf. Section 4.2, p. 99), where a system either has just one indicator or each of its parameters only affects a single indicator, may be insufficient for some systems depending on their internal complexity or the level of environmental uncertainty. The complexity in this situation is due to parameters possibly influencing many indicators, which gives rise to trade-offs.

Trade-off aversion is natural [Zeleny, 2010] and there are ways to avoid them. One possibility is to come up with a single objective function to evaluate alternatives, which is built given the information about priorities of indicators (e.g., using the appropriate weights). However, due to the incompleteness of information about how parameters affect indicators, such function is not an option for us. There are two ways we could deal with multiple indicators:

- Create new procedures for the existing process of *Qualia*, depicted in Figure 4.4 (p. 102), which consider the negative side effects on other indicators, but does not support handling multiple indicators concurrently;
- Extend the current process of *Qualia* with a new activity capable of dealing with multiple concurrent indicator failures.

In either case, *Qualia* requires an additional piece of information in the specification: the priorities of the indicators (e.g., [Liaskos et al., 2011, Section 3.3]). Thus, in the context of multiple indicators, the aim of the process is to maintain indicators satisfied according to their priority: while it prefers to keep most indicators satisfied, if a satisfaction of a higher-priority indicator implies a failure of a lower-priority one, this is generally acceptable.

We are also studying the applicability of two-phase locking from databases/transaction processing to the scenario of multiple concurrent failures. The idea here is to place locks on indicators that have failed and are currently being worked on, preventing other failures to change parameters that would affect the locked indicator in a certain way.

Bibliography

Ali, Raian; Chopra, Amit K.; Dalpiaz, Fabiano; Giorgini, Paolo; Mylopoulos, John, and Souza, Vítor E. S. The Evolution of Tropos: Contexts, Commitments and Adaptivity. In *Proc. of the 4th International i* Workshop*, pages 15–19, 2010a.

Ali, Raian; Dalpiaz, Fabiano, and Giorgini, Paolo. A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, 15(4):439–458, 2010b.

Ali, Raian; Dalpiaz, Fabiano; Giorgini, Paolo, and Souza, Vítor E. S. Requirements Evolution: From Assumptions to Reality. In Halpin, Terry; Nurcan, Selmin; Krogstie, John; Soffer, Pnina; Proper, Erik; Schmidt, Rainer, and Bider, Ilia, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 81 of *Lecture Notes in Business Information Processing*, pages 372–382. Springer, 2011a.

Ali, Raian; Solis, Carlos; Omoronyia, Inah; Salehie, Mazeiar, and Nuseibeh, Bashar. Social Adaptation - When Software Gives Users a Voice. Technical Report November, Lero-TR-2011-05, University of Limerick, Ireland, 2011b.

Andersson, Jesper; de Lemos, Rogério; Malek, Sam, and Weyns, Danny. Modeling Dimensions of Self-Adaptive Software Systems. In Cheng, Betty H. C.; de Lemos, Rogério; Giese, Holger; Inverardi, Paola, and Magee, Jeff, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 27–47. Springer, 2009.

Antón, Annie I. Goal-Based Requirements Analysis. In *Proc. of the 2nd International Conference on Requirements Engineering*, pages 136–144. IEEE, 1996.

Antón, Annie I. and Potts, Colin. Functional Paleontology: System Evolution as the User Sees It. In *Proc. of the 23rd International Conference on Software Engineering*, pages 421–430. IEEE, 2001.

Baresi, Luciano and Pasquale, Liliana. Adaptive Goals for Self-Adaptive Service Compositions. In *Proc. of the 2010 IEEE International Conference on Web Services*, pages 353–360. IEEE, 2010.

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

Ben Charrada, Eya and Glinz, Martin. An Automated Hint Generation Approach for Supporting the Evolution of Requirements Specifications. In *Proc. of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 58–62. ACM, 2010.

Bencomo, Nelly; Blair, Gordon; Cheng, Betty H. C.; France, Robert, and Jeanneret, Cédric, editors. *Proceedings of the 6th International Workshop on Models@run.time at the ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*. CEUR, 2011a.

Bencomo, Nelly; Letier, Emmanuel; Finkelstein, Anthony; Whittle, Jon, and Welsh, Kriss, editors. *Proceedings of the 2nd International Workshop on Requirements@Run.Time*. IEEE, 2011b.

Berry, Daniel M.; Cheng, Betty H. C., and Zhang, Ji. The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems. In *Proc. of the 11th International Workshop on Requirements Engineering: Foundation for Software Quality*, pages 95–100, 2005.

Beynon-Davies, Paul. Information systems 'failure': the case of the London Ambulance Service's Computer Aided Despatch project. *European Journal of Information Systems*, 4(3):171–184, 1995.

Brake, Nevon; Cordy, James R.; Dancy, Elizabeth; Litoiu, Marin, and Popescu, Valentina. Automating Discovery of Software Tuning Parameters. In *Proc. of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 65–72. ACM, 2008.

Breitman, Karin K.; Leite, Julio C. S. P., and Finkelstein, Anthony. The world's a stage: a survey on requirements engineering using a real-life case study. *Journal of the Brazilian Computer Society*, 6(1), 1999.

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

Brown, Greg; Cheng, Betty H. C.; Goldsby, Heather, and Zhang, Ji. Goal-oriented Specification of Adaptation Requirements Engineering in Adaptive Systems. In *Proc. of the 2006 International Workshop on Self-adaptation and Self-managing Systems*, pages 23–29. ACM, 2006.

Brueckner, Sven and Geihs, Kurt, editors. *Proceedings of the 5th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 2011.

Brun, Yuriy and others, . Engineering Self-Adaptive Systems through Feedback Loops. In Cheng, Betty H. C.; de Lemos, Rogério; Giese, Holger; Inverardi, Paola, and Magee, Jeff, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer, 2009.

Bryl, Volha. *Supporting the Design of Socio-Technical Systems by Exploring and Evaluating Design Alternatives*. Phd thesis, University of Trento, 2009.

Butler, Basil R. R. and others, . The Challenges of Complex IT Projects. Technical report, The Royal Academy of Engineering (available at: `http://www.raeng.org.uk/ComplexIT`), 2004.

Castro, Jaelson; Kolp, Manuel, and Mylopoulos, John. Towards requirements-driven information systems engineering: the Tropos project. *Information Systems*, 27(6):365–389, 2002.

Chelf, Ben and Chou, Andy. Controlling Software Complexity: The Business Case for Static Source Code Analysis. Technical report, Coverity, Inc. (available at: `http://www.coverity.com/library/pdf/ControllingSoftwareComplexity.pdf`), 2008.

Cheng, Betty H. C. and Atlee, Joanne M. Research Directions in Requirements Engineering. In *Future of Software Engineering (FOSE '07)*, pages 285–303. IEEE, 2007.

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

Cheng, Betty H. C. and others, . Software Engineering for Self-Adaptive Systems: A Research Roadmap. In Cheng, Betty H. C.; de Lemos, Rogério; Giese, Holger; Inverardi, Paola, and Magee, Jeff, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009b.

Cheng, Shang-Wen. *Rainbow: Cost-Effective Software Architecture-based Self-adaptation*. PhD thesis, Carnegie Mellon University, 2008.

Cheng, Shang-Wen; Garlan, David, and Schmerl, Bradley. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *Proc. of the ICSE 2009 Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141. IEEE, 2009c.

Cornford, Steven L.; Feather, Martin S., and Hicks, Kenneth A. DDP: A Tool for Life-Cycle Risk Management. *IEEE Aerospace and Electronic Systems Magazine*, 21(6):13–22, 2006.

Dalpiaz, Fabiano. *Exploiting Contextual and Social Variability for Software Adaptation*. Phd thesis, University of Trento, 2011.

Dalpiaz, Fabiano; Giorgini, Paolo, and Mylopoulos, John. An Architecture for Requirements-Driven Self-reconfiguration. In van Eck, Pascal; Gordijn, Jaap, and Wieringa, Roel, editors, *Advanced Information Systems Engineering*, volume 5565 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2009.

Dalpiaz, Fabiano; Chopra, Amit K.; Giorgini, Paolo, and Mylopoulos, John. Adaptation in Open Systems: Giving Interaction Its Rightful Place. In Parsons, Jeffrey; Saeki, Motoshi; Shoval, Peretz; Woo, Carson, and Wand, Yair, editors, *Conceptual Modeling – ER 2010*, volume 6412 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2010.

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

Dardenne, Anne; van Lamsweerde, Axel, and Fickas, Stephen. Goal-directed Requirements Acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

Dastani, Mehdi; van Riemsdijk, M. Birna, and Meyer, John-Jules C. Goal Types in Agent Programming. In *Proc. of the 5$^{\text{th}}$ International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1285–1287. ACM, 2006.

Di Nitto, Elisabetta; Ghezzi, Carlo; Metzger, Andreas; Papazoglou, Mike, and Pohl, Klaus. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3):313–341, 2008.

Doyle, John C.; Francis, Bruce A., and Tannenbaum, Allen R. *Feedback Control Theory*. Macmillan Coll Div, 1992 edition, 1992.

Duan, Jinan. Revolution2: A Tool for Enterprise Application Evolution Based on Requirement-Driven Approach. In *Proc. of the 2009 International Conference on Information Engineering and Computer Science*, pages 1–4. IEEE, 2009.

Dwyer, Matthew B.; Avrunin, George S., and Corbett, James C. Patterns in Property Specifications for Finite-State Verification. In *Proc. of the 21$^{\text{st}}$ International Conference on Software Engineering*, pages 411–420. ACM, 1999.

Easterbrook, Steve and Aranda, Jorge. Case Studies for Software Engineers, tutorial presented at the 28$^{\text{th}}$ International Conference on Software Engineering, 2006.

Ebert, Christof and De Man, Jozef. Requirements Uncertainty: Influencing Factors and Concrete Improvements. In *Proc. of the 27th International Conference on Software Engineering*, pages 553–560. ACM, 2005.

Elahi, Golnaz and Yu, Eric S. K. Requirements Trade-offs Analysis in the Absence of Quantitative Measures: A Heuristic Method. In *Proc. of the 2011 ACM Symposium on Applied Computing*, pages 651–658. ACM, 2011.

Ernst, Neil A.; Mylopoulos, John, and Wang, Yiqiao. Requirements Evolution and What (Research) to Do about It. In Lyytinen, Kalle; Loucopoulos, Pericles; Mylopoulos, John, and Robinson, Bill, editors, *Design Requirements Engineering: A Ten-Year Perspective*, pages 186–214. Springer, 2009.

Ernst, Neil A.; Borgida, Alex, and Jureta, Ivan. Finding Incremental Solutions for Evolving Requirements. In *Proc. of the 19th International Requirements Engineering Conference*, pages 15–24. IEEE, 2011.

Feather, M.S.; Fickas, Stephen; van Lamsweerde, Axel, and Ponsard, Christophe. Reconciling system requirements and runtime behavior. In *Proc. of the 9th International Workshop on Software Specification and Design*, pages 50–59. IEEE, 1998.

Fickas, Stephen and Feather, Martin S. Requirements Monitoring in Dynamic Environments. In *Proc. of the 2nd IEEE International Symposium on Requirements Engineering*, pages 140–147. IEEE, 1995.

Filieri, Antonio; Ghezzi, Carlo; Leva, Alberto, and Maggio, Martina. Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements. In *Proc. of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 283–292. IEEE, 2011.

Filieri, Antonio; Ghezzi, Carlo; Leva, Alberto, and Maggio, Martina. Reliability-driven dynamic binding via feedback control. In *Private communication*, 2012.

Finkelstein, Anthony. Report of the inquiry into the London Ambulance Service (Electronic Version). Technical report, South West Thames Regional Health Authority, 1993.

Finkelstein, Anthony and Dowell, John. A Comedy of Errors: the London Ambulance Service case study. In *Proc. of the 8th International Workshop on Software Specification and Design*, pages 2–4. IEEE, 1996.

Flake, Stephan. Enhancing the Message Concept of the Object Constraint Language. In *Proc. of the 16th International Conference on Software Engineering and Knowledge Engineering*, pages 161–166, 2004.

Forbus, Kenneth D. Qualitative Reasoning. In *Computer Science Handbook*, chapter 62. Chapman and Hall/CRC, 2nd edition, 2004.

Foster, Howard; Uchitel, Sebastian; Kramer, Jeff, and Magee, Jeff. Towards Self-management in Service-Oriented Computing with Modes. In Di Nitto, Elisabetta and Ripeanu, Matei, editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, volume 4907 of *Lecture Notes in Computer Science*, pages 338–350. Springer, 2009.

Fu, Lingxiao; Peng, Xin; Yu, Yijun, and Zhao, Wenyun. Stateful Requirements Monitoring for Self-Repairing of Software Systems. Technical report, FDSE-TR201101 (available online: `http://www.se.fudan.sh.cn/paper/techreport/1.pdf`), Fudan University, China, 2010.

Ganek, Alan G. and Corbi, Thomas A. The dawning of the autonomic computing era. *IBM Systems Journal*, 42 (1):5–18, 2003.

Garcia-Molina, Hector and Salem, Kenneth. Sagas. *ACM SIGMOD Record*, 16(3):249–259, 1987.

Garlan, David; Siewiorek, Dan P.; Smailagic, Asim, and Steenkiste, Peter. Project Aura: toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

Garlan, David; Cheng, Shang-Wen, and Schmerl, Bradley. Increasing System Dependability through Architecture-Based Self-Repair. In de Lemos, Rogério; Gacek, Cristina, and Romanovsky, Alexander, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 61–89. Springer, 2003.

Garlan, David; Cheng, Shang-Wen; Huang, An-Cheng; Schmerl, Bradley, and Steenkiste, Peter. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.

Gat, Erann. On Three-Layer Architectures. In Kortenkamp, David; Bonasso, R. Peter, and Murphy, Robin, editors, *Artificial Intelligence and Mobile Robots*, pages 195–210. MIT Press, 1998.

Gell-Mann, Murray. Simplicity and Complexity in the Description of Nature. *Engineering and Science*, 57(3): 2–9, 1988.

Georgiadis, Ioannis; Magee, Jeff, and Kramer, Jeff. Self-Organising Software Architectures for Distributed Systems. In *Proc. of the 1st Workshop on Self-healing Systems*, pages 33–38. ACM, 2002.

Giorgini, Paolo; Mylopoulos, John; Nicchiarelli, Eleonora, and Sebastiani, Roberto. Reasoning with Goal Models. In Spaccapietra, Stefano; March, Salvatore, and Kambayashi, Yahiko, editors, *Conceptual Modeling – ER 2002*, volume 2503 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2003.

Giorgini, Paolo; Mylopoulos, John, and Sebastiani, Roberto. Goal-oriented requirements analysis and reasoning in the Tropos methodology. *Engineering Applications of Artificial Intelligence*, 18(2):159–171, 2005.

Giunchiglia, Fausto; Mylopoulos, John, and Perini, Anna. The Tropos Software Development Methodology: Processes, Models and Diagrams. *Agent-Oriented Software Engineering III*, 2585:162–173, 2003.

Goldsby, Heather J.; Sawyer, Pete; Bencomo, Nelly; Cheng, Betty H. C., and Hughes, Danny. 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.

Gonzalez-Baixauli, Bruno; Sampaio do Prado Leite, J.C., and Mylopoulos, John. Visual Variability Analysis for Goal Models. In *Proc. of the 12th IEEE International Requirements Engineering Conference*, pages 183–192. IEEE, 2004.

Griss, Martin L.; Favaro, John, and D'Alessandro, Massimo. Integrating feature modeling with the RSEB. In *Proc. of the 5th International Conference on Software Reuse*, pages 76–85. IEEE, 1998.

Harker, Susan D.P.; Eason, Ken D., and Dobson, John E. The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering. In *Proc. of the 1993 IEEE International Symposium on Requirements Engineering*, pages 266–272. IEEE, 1993.

Hawthorne, Matthew J. and Perry, Dewayne E. Exploiting Architectural Prescriptions for Self-Managing, Self-Adaptive Systems: A Position Paper. In *Proc. of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, pages 75–79. ACM, 2004.

Heaven, William and Letier, Emmanuel. Simulating and Optimising Design Decisions in Quantitative Goal Models. In *Proc. of the 19th IEEE International Requirements Engineering Conference*, pages 79–88. IEEE, 2011.

Heaven, William; Sykes, Daniel; Magee, Jeff, and Kramer, Jeff. A Case Study in Goal-Driven Architectural Adaptation. In Cheng, Betty H. C.; de Lemos, Rogério; Giese, Holger; Inverardi, Paola, and Magee, Jeff, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2009.

Hebig, Regina; Giese, Holger, and Becker, Basil. Making Control Loops Explicit when Architecting Self-Adaptive Systems. In *Proc. of the 2nd International Workshop on Self-organizing Architectures*, pages 21–28. ACM, 2010.

Hellerstein, Joseph L.; Diao, Yixin; Parekh, Sujay, and Tilbury, Dawn M. *Feedback Control of Computing Systems*. Wiley, 1st edition, 2004.

Hevner, Alan R.; March, Salvatore T.; Park, Jinsoo, and Ram, Sudha. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

Hinchey, Mike and Coyle, Lorcan. *Conquering Complexity*. Springer, 2012.

Hong, Dan; Chiu, Dickson K. W., and Shen, Vincent Y. Requirements Elicitation for the Design of Context-aware Applications in a Ubiquitous Environment. In *Proc. of the 7th International Conference on Electronic Commerce*, pages 590–596. ACM, 2005.

Horn, Paul. Autonomic Computing: IBM' s Perspective on the State of Information Technology, `http://www.research.ibm.com/autonomic/manifesto/` (last access: March 20th, 2012), 2001.

Huebscher, Markus C. and McCann, Julie A. A survey of Autonomic ComputingDegrees, Models, and Applications. *ACM Computing Surveys*, 40(3):1–28, 2008.

Jureta, Ivan; Mylopoulos, John, and Faulkner, Stephane. Revisiting the Core Ontology and Problem in Requirements Engineering. In *Proc. of the 16th IEEE International Requirements Engineering Conference*, pages 71–80. IEEE, 2008.

Kaindl, Hermann. A practical approach to combining requirements definition and object-oriented analysis. *Annals of Software Engineering*, 3(1):319–343, 1997.

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

Khan, Malik J.; Awais, Mian M., and Shamail, Shafay. Enabling Self-Configuration in Autonomic Systems using Case-Based Reasoning with Improved Efficiency. In *Proc. of the 4th International Conference on Autonomic and Autonomous Systems*, pages 112–117. IEEE, 2008.

Kramer, Jeff and Magee, Jeff. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268. IEEE, 2007.

Kramer, Jeff and Wolf, Alexander L. Succeedings of the 8th International Workshop on Software Specification and Design. *ACM SIGSOFT Software Engineering Notes*, 21(5):21–35, 1996.

Kuipers, Benjamin. Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge. *Automatica*, 25(4):571–585, 1989.

Laddaga, Robert and Robertson, Paul. Self Adaptive Software: A Position Paper. In *Proc. of the 2004 International Workshop on Self-* Properties in Complex Information Systems*, 2004.

Lam, Wai and Loomes, Martin. Requirements Evolution in the Midst of Environmental Change: a Managed Approach. In *Proc. of the 2nd Euromicro Conference on Software Maintenance and Reengineering*, pages 121–127. IEEE, 1998.

Lapouchnian, Alexei. Goal-Oriented Requirements Engineering: An Overview of the Current Research. Technical report, University of Toronto, Canada (available online: `http://www.cs.toronto.edu/~alexei/pub/Lapouchnian-Depth.pdf`), 2005.

Lapouchnian, Alexei. *Exploiting Requirements Variability for Software Customization and Adaptation*. Phd thesis, University of Toronto, Canada, 2010.

Lapouchnian, Alexei and Mylopoulos, John. Modeling Domain Variability in Requirements Engineering with Contexts. In Laender, Alberto; Castano, Silvana; Dayal, Umeshwar; Casati, Fabio, and de Oliveira, José, editors, *Conceptual Modeling - ER 2009*, volume 5829 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2009.

Lehman, Meir M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

Lehman, Meir M. and Ramil, Juan F. Software Uncertainty. In Bustard, David; Liu, Weiru, and Sterritt, Roy, editors, *Soft-Ware 2002: Computing in an Imperfect World*, volume 2311 of *Lecture Notes in Computer Science*, pages 477–514. Springer, 2002.

Letier, Emmanuel. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. Phd thesis, Université Catholique de Louvain, Belgium, 2001.

Letier, Emmanuel and van Lamsweerde, Axel. Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering. In *Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, volume 29, pages 53–62. ACM, 2004.

Liaskos, Sotirios; Lapouchnian, Alexei; Yu, Yijun; Yu, Eric S. K., and Mylopoulos, John. On Goal-based Variability Acquisition and Analysis. In *Proc. of the 14th IEEE International Requirements Engineering Conference*, pages 79–88. IEEE, 2006.

Liaskos, Sotirios; McIlraith, Sheila; Sohrabi, Shirin, and Mylopoulos, John. Representing and reasoning about preferences in requirements engineering. *Requirements Engineering*, 16(3):227–249, 2011.

Ma, Wenting; Liu, Lin; Xie, Haihua; Zhang, Hongyu, and Yin, Jinglei. Preference Model Driven Services Selection. In van Eck, Pascal; Gordijn, Jaap, and Wieringa, Roel, editors, *Advanced Information Systems Engineering*, volume 5565 of *Lecture Notes in Computer Science*, pages 216–230. Springer, 2009.

Menasce, Daniel A.; Gomaa, Hassan; Malek, Sam, and Sousa, João P. SASSY: A Framework for Self-Architecting Service-Oriented Systems. *IEEE Software*, 28(6):78–85, 2011.

Menzies, Tim and Richardson, Julian. Qualitative Modeling for Requirements Engineering. In *Proc. of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 11–20. IEEE, 2006.

Moody, D. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.

Morandini, Mirko. *Goal-Oriented Development of Self-Adaptive Systems*. Phd thesis, University of Trento, 2011.

Morandini, Mirko; Penserini, Loris, and Perini, Anna. Towards Goal-Oriented Development of Self-Adaptive Systems. In *Proc. of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, pages 9–16. ACM, 2008.

Morandini, Mirko; Penserini, Loris, and Perini, Anna. 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.

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

Mylopoulos, John; Chung, Lawrence, and Yu, Eric S. K. From Object-Oriented to Goal-Oriented Requirements Analysis. *Communications of the ACM*, 42(1):31–37, 1999.

Nakagawa, Hiroyuki; Ohsuga, Akihiko, and Honiden, Shinichi. gocc: A Configuration Compiler for Self-adaptive Systems Using Goal-oriented Requirements Description. In *Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 40–49. ACM, 2011.

Nissen, Hans W.; Schmitz, Dominik; Jarke, Matthias; Rose, Thomas; Drews, Peter; Hesseler, Frank J., and Reke, Michael. Evolution in Domain Model-Based Requirements Engineering for Control Systems Development. In *Proc. of the 17th IEEE International Requirements Engineering Conference*, pages 323–328. IEEE, 2009.

Northrop, Linda and others, . *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon, 2006.

Oreizy, Peyman and others, . An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

Pasquale, Liliana. *A Goal-oriented Methodology for Self-supervised Service Compositions*. Phd thesis, Politecnico di Milano, 2010.

Peng, Xin; Chen, Bihuan; Yu, Yijun, and Zhao, Wenyun. Self-Tuning of Software Systems through Goal-based Feedback Loop Control. In *Proc. of the 18th IEEE International Requirements Engineering Conference*, pages 104–107. IEEE, 2010.

Qureshi, Nauman; Jureta, Ivan, and Perini, Anna. Requirements Engineering for Self-Adaptive Systems: Core Ontology and Problem Statement. In Mouratidis, Haralambos and Rolland, Colette, editors, *Advanced Information Systems Engineering*, volume 6741 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin / Heidelberg, 2011a.

Qureshi, Nauman A. and Perini, Anna. Engineering Adaptive Requirements. In *Proc. of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 126–131. IEEE, 2009.

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

Qureshi, Nauman A.; Liaskos, Sotirios, and Perini, Anna. Reasoning About Adaptive Requirements for Self-Adaptive Systems at Runtime. In *Proc. of the 2nd International Workshop on Requirements@Run.Time*, pages 16–22. IEEE, 2011b.

Ramirez, Andres J. and Cheng, Betty H. C. Design Patterns for Developing Dynamically Adaptive Systems. In *Proc. of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 49–58. ACM, 2010.

Ritsko, John J., editor. *IBM Systems Journal, Volume 42, Issue 1.* IBM, 2003.

Robinson, William N. Implementing Rule-Based Monitors within a Framework for Continuous Requirements Monitoring. In *Proc. of the 38th Annual Hawaii International Conference on System Sciences*, page 188a. IEEE, 2005.

Robinson, William N. A requirements monitoring framework for enterprise systems. *Requirements Engineering*, 11(1):17–41, 2006.

Robinson, William N. Extended OCL for Goal Monitoring. *Electronic Communications of the EASST*, 9, 2008.

Robinson, William N. and Fickas, Stephen. Designs Can Talk: A Case of Feedback for Design Evolution in Assistive Technology. In Lyytinen, Kalle and others, , editors, *Design Requirements Engineering: A Ten-Year Perspective*, volume 14 of *Lecture Notes in Business Information Processing*, pages 215–237. Springer, 2009.

Robinson, William N. and Purao, Sandeep. Monitoring Service Systems from a Language-Action Perspective. *IEEE Transactions on Services Computing*, 4(1):17–30, 2011.

Rosenthal, David. *Consciousness and Mind.* Oxford University Press, 1st edition, 2005.

Ross, Douglas T. and Schoman Jr., Kenneth E. Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, SE-3(1):6–15, 1977.

Salehie, Mazeiar and Tahvildari, Ladan. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, 2009.

Salehie, Mazeiar and Tahvildari, Ladan. Towards a Goal-Driven Approach to Action Selection in Self-Adaptive Software. *Software: Practice and Experience*, 42(2):211–233, 2012.

Salifu, Mohammed; Nuseibeh, Bashar; Rapanotti, Lucia, and Tun, Than T. Using Problem Descriptions to Represent Variability for Context-Aware Applications. In *Proc. of the 1st International Workshop on Variability Modelling of Software-intensive Systems*, page (available online: `http://oro.open.ac.uk/id/eprint`. The Open University, UK, 2007.

Sawyer, Pete; Bencomo, Nelly; Whittle, Jon; Letier, Emmanuel, and Finkelstein, Anthony. Requirements-Aware Systems: A research agenda for RE for self-adaptive systems. In *Proc. of the 18th IEEE International Requirements Engineering Conference*, pages 95–103. IEEE, 2010.

Schaad, Andreas. TAM2: Automated Threat Analysis, Oral presentation during SE-1 session at the 27th Symposium On Applied Computing, 2012.

Schmerl, Bradley and Garlan, David. Exploiting Architectural Design Knowledge to Support Self-Repairing Systems. In *Proc. of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 241–248. ACM, 2002.

Schmidt, Kjeld. The Problem with 'Awareness': Introductory Remarks on 'Awareness in CSCW'. *Computer Supported Cooperative Work (CSCW)*, 11(3):285–298, 2002.

Schmitz, Dominik; Nissen, Hans W.; Jarke, Matthias; Rose, Thomas; Drews, Peter; Hesseler, Frank J., and Reke, Michael. Requirements Engineering for Control Systems Development in Small and Medium-Sized Enterprises. In *Proc. of the 16th IEEE International Requirements Engineering Conference*, pages 229–234. IEEE, 2008.

Schmitz, Dominik and others, . Mapping Requirement Models to Mathematical Models in Control System Development. In Paige, Richard; Hartman, Alan, and Rensink, Arend, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 2009.

Semmak, Farida; Gnaho, Christophe, and Laleau, Régine. Extended Kaos to Support Variability for Goal Oriented Requirements Reuse. In Ebersold, Sophie; Front, Agnès; Lopistéguy, Philippe, and Nurcan, Selmin, editors, *Proc. of the International Workshop on Model Driven Information Systems Engineering: Enterprise, User and System Models*, pages 22–33. CEUR, 2008.

Sillitti, A.; Ceschi, M.; Russo, B., and Succi, G. Managing Uncertainty in Requirements: A Survey in Documentation-Driven and Agile Companies. In *Proc. of the 11$^{\text{th}}$ IEEE International Software Metrics Symposium*, pages 17–26. IEEE, 2005.

Sousa, João P.; Balan, Rajesh K.; Poladian, Vahe; Garlan, David, and Satyanarayanan, Mahadev. A Software Infrastructure for UserGuided QualityofService Tradeoffs. In Cordeiro, José; Shishkov, Boris; Ranchordas, AlpeshKumar, and Helfert, Markus, editors, *Software and Data Technologies*, volume 47 of *Communications in Computer and Information Science*, pages 48–61. Springer, 2009.

Souza, Vítor E. S. Conceptual Modeling 2010 Assignment Report. Technical report, University of Trento, Italy, 2010.

Souza, Vítor E. S. An Experiment on the Development of an Adaptive System based on the LAS-CAD. Technical report, University of Trento (available at: `http://disi.unitn.it/~vitorsouza/a-cad/`), 2012.

Souza, Vítor E. S. and Mylopoulos, John. Monitoring and Diagnosing Malicious Attacks with Autonomic Software. In Laender, Alberto; Castano, Silvana; Dayal, Umeshwar; Casati, Fabio, and de Oliveira, José, editors, *Conceptual Modeling - ER 2009*, volume 5829 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.

Souza, Vítor E. S. and Mylopoulos, John. From Awareness Requirements to Adaptive Systems: a Control-Theoretic Approach. In *Proc. of the 2$^{\text{nd}}$ International Workshop on Requirements@Run.Time*, pages 9–15. IEEE, 2011.

Souza, Vítor E. S.; Lapouchnian, Alexei, and Mylopoulos, John. System Identification for Adaptive Software Systems: a Requirements Engineering Perspective. In Jeusfeld, Manfred; Delcambre, Lois, and Ling, Tok-Wang, editors, *Conceptual Modeling – ER 2011*, volume 6998 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2011a.

Souza, Vítor E. S.; Lapouchnian, Alexei; Robinson, William N., and Mylopoulos, John. Awareness Requirements for Adaptive Systems. In *Proc. of the 6$^{\text{th}}$ International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 60–69. ACM, 2011b.

Souza, Vítor E. S.; Lapouchnian, Alexei; Angelopoulos, Konstantinos, and Mylopoulos, John. Requirements-Driven Software Requirements Evolution. *Computer Science - Research and Development (to appear)*, 2012a.

Souza, Vítor E. S.; Lapouchnian, Alexei, and Mylopoulos, John. Requirements-driven Qualitative Adaptation. In *Proc. of the 20th International Conference on Cooperative Information Systems (to appear)*. Springer, 2012b.

Souza, Vítor E. S.; Lapouchnian, Alexei, and Mylopoulos, John. System Identification for Requirements-Driven Qualitative Runtime Adaptation. *LNCS State-of-the-Art Survey Volume on Models@run.time (to appear)*, 2012c.

Souza, Vítor E. S.; Lapouchnian, Alexei, and Mylopoulos, John. (Requirement) Evolution Requirements for Adaptive Systems. In *Proc. of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (to appear)*, 2012d.

Souza, Vítor E. S.; Lapouchnian, Alexei; Robinson, William N., and Mylopoulos, John. Awareness Requirements for Adaptive Systems. In de Lemos, Rogério; Giese, Holger; Müller, Hausi A., and Shaw, Mary, editors, *Software Engineering for Self-Adaptive Systems 2 (to appear)*. Springer, 2012e.

Souza, Vítor E. S.; Mazón, Jose-Norberto; Garrigós, Irene; Trujillo, Juan, and Mylopoulos, John. Monitoring Strategic Goals in Data Warehouses with Awareness Requirements. In *Proc. of the 2012 ACM Symposium on Applied Computing*. ACM, 2012f.

Sykes, Daniel; Heaven, William; Magee, Jeff, and Kramer, Jeff. Plan-Directed Architectural Change For Autonomous Systems. In *Proc. of the 2007 Conference on Specification and Verification of Component-based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–21. ACM, 2007.

Sykes, Daniel; Heaven, William; Magee, Jeff, and Kramer, Jeff. From Goals To Components: A Combined Approach To Self-Management. In *Proc. of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, pages 1–8. ACM, 2008.

Sykes, Daniel; Heaven, William; Magee, Jeff, and Kramer, Jeff. Exploiting Non-Functional Preferences in Architectural Adaptation for Self-Managed Systems. In *Proc. of the 2010 ACM Symposium on Applied Computing*, pages 431–438. ACM, 2010.

Tallabaci, Genci. *System Identification for the ATM System*. Master thesis, University of Trento (to be submitted), 2012.

Taylor, Richard and Tofts, Chris. Self Managed Systems - A Control Theory Perspective. Technical report, HPL-2004-49, HP Laboratories Bristol, 2004.

van Lamsweerde, Axel. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proc. of the 5th IEEE International Symposium on Requirements Engineering*, pages 249–262. IEEE, 2001.

van Lamsweerde, Axel. Reasoning About Alternative Requirements Options. In Borgida, Alexander T.; Chaudhri, Vinay K.; Giorgini, Paolo, and Yu, Eric S. K., editors, *Conceptual Modeling: Foundations and Applications*, chapter 20, pages 380–397. Springer, 2009.

van Lamsweerde, Axel and Letier, Emmanuel. From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering. In *Proc. of the 9th International Workshop on Radical Innovations of Software and Systems Engineering in the Future*, pages 4–8. Springer, 2002.

van Lamsweerde, Axel and Willemet, Laurent. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, 1998.

van Lamsweerde, Axel; Dardenne, Anne; Delcourt, B., and Dubisy, F. The KAOS Project: Knowledge Acquisition in Automated Specification of Software. In *Proc. of the AAAI Spring Symposium Series, Stanford University*, pages 59–62. AAAI, 1991.

Villela, Karina; Doerr, Joerg, and Gross, Anne. Proactively Managing the Evolution of Embedded System Requirements. In *Proc. of the 16th IEEE International Requirements Engineering Conference*, pages 13–22. IEEE, 2008.

Wang, Yiqiao and Mylopoulos, John. Self-Repair through Reconfiguration: A Requirements Engineering Approach. In *Proc. of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 257–268. IEEE, 2009.

Welsh, Kristopher; Sawyer, Pete, and Bencomo, Nelly. Run-time Resolution of Uncertainty. In *Proc. of the 19th IEEE International Requirements Engineering Conference*, pages 355–356. IEEE, 2011.

Wen-jie, Yuan and Shi, Ying. Evolution-Oriented Reflective Requirements Specifications and its Formalization. In *Proc. of the 3rd International Symposium on Intelligent Information Technology Application*, volume 3, pages 164–167. IEEE, 2009.

Whittle, Jon; Sawyer, Pete; Bencomo, Nelly; Cheng, Betty H. C., and Bruel, Jean-Michel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *Proc. of the 17th IEEE International Requirements Engineering Conference*, pages 79–88. IEEE, 2009.

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

Wohlin, Claes; Runeson, Per, and Höst, Martin. *Experimentation in Software Engineering: An Introduction*. 1999.

You, Zheng. Experiences with applying the i* framework to a real-life system. Technical report, Requirements Engineering (CSC2106) Course Project, University of Toronto, Canada, 2001.

You, Zheng. *Using Meta-Model-Driven Views to Address Scalability in i* Models*. Master thesis, University of Toronto, Canada, 2004.

Yu, Eric S. K. Social Modeling and i*. In Borgida, Alex; Chaudhri, Vinay; Giorgini, Paolo, and Yu, Eric, editors, *Conceptual Modeling: Foundations and Applications*, chapter 7, pages 99–121. Springer, 2009.

Yu, Eric S. K. and Mylopoulos, John. Understanding "Why" in Software Process Modelling, Analysis, and Design. In *Proc. of the 16th International Conference on Software Engineering*, pages 159–168. IEEE, 1994.

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

Zave, Pamela and Jackson, Michael. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.

Zeleny, Milan. Multiobjective Optimization, Systems Design and De Novo Programming. In Zopounidis, Constantin; Pardalos, Panos M., and Hearn, Donald W, editors, *Handbook of Multicriteria Analysis*, volume 103 of *Applied Optimization*, chapter 8, pages 243–262. Springer, 2010.

Zhang, Ji and Cheng, Betty H. C. Specifying Adaptation Semantics. In *Proc. of the 2005 Workshop on Architecting Dependable Systems*, pages 1–7. ACM, 2005.

Zhang, Ji and Cheng, Betty H. C. Model-Based Development of Dynamically Adaptive Software. In *Proc. of the 28th International Conference on Software Engineering*, pages 371–380. ACM, 2006.

Zhu, Xiaoyun; Uysal, Mustafa; Wang, Zhikui; Singhal, Sharad; Merchant, Arif; Padala, Pradeep, and Shin, Kang. What Does Control Theory Bring to Systems Research? *ACM SIGOPS Operating Systems Review*, 43(1):62, 2009.

Zowghi, Didar and Offen, Ray. A Logical Framework for Modeling and Reasoning about the Evolution of Requirements. In *Proc. of the 3rd IEEE International Symposium on Requirements Engineering*, pages 247–257. IEEE, 1997.