PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Exploiting Contextual and Social Variability for Software Adaptation

Fabiano Dalpiaz

Advisor:

Prof. John Mylopoulos

Università degli Studi di Trento

Co-Advisor:

Prof. Paolo Giorgini

Università degli Studi di Trento

January 2011

Abstract

Self-adaptive software systems are systems that monitor their environment and compensate if there are deviations from their requirements. Self-adaptivity is gaining prominence as an approach to lowering software costs by reducing the need for manual system maintenance. Self-adaptivity is particularly important for distributed systems that involve both software and human/organizational actors because of the volatility as well as uncertainty that permeates their operational environments. We refer to such systems as Socio-Technical System (STS).

The thesis proposes a comprehensive framework for designing self-adaptive software that operates within a socio-technical system. The framework is founded upon the notions of contextual and social variability. A key ingredient of our approach is to rely on high-level abstractions to represent the purpose of the system (requirements model), to explicitly represent the commitments that exist among participating actors in an STS, and also to consider how operational context influences requirements. The proposed framework consists of (i) modelling and analysis techniques for representing and reasoning about contextual and social variability; (ii) a conceptual architecture for self-adaptive STSs; and (iii) a set of algorithms to diagnose a failure and to compute and select a new variant that addresses the failure. To evaluate our proposal, we developed two prototype implementations of our architecture to demonstrate different features of our framework, and successfully applied them to two case studies. In addition, the thesis reports encouraging results on experiments we conducted with our implementations in order to check for scalability.

Keywords
[Adaptive Software, Requirements Engineering, Multi-Agent Systems, Variability, Socio-Technical Systems]

Acknowledgements

I am extremely grateful to John Mylopoulos, whose vision and enthusiasm have guided and encouraged my research over the last four years. It was a privilege to have such an outstanding supervisor. I owe special thanks to Paolo Giorgini, not only for his co-supervision, but also for convincing me to leave my job in industry and embark on a doctorate. Thanks to the professors who served on my thesis committee: Luciano Baresi (Politecnico di Milano), Schahram Dustdar (Technical University of Vienna), and Franco Zambonelli (Università di Modena e Reggio Emilia).

I am thankful to my main co-authors Raian Ali and Amit K. Chopra. Each played an essential role in different phases of my thesis. I will always remember with pleasure our intellectually stimulating meetings. I can't help saying "Grazie mille!" to Michele Vescovi, who developed the automated reasoner $\mathcal{EL}^{+}$2SAT that I applied to the variants generation problem. You all are very good friends of mine.

My discussions with professors and researchers have stimulated my work and have provided me with useful directions and suggestions. I benefited much from the discussions with researchers located in Trento, in particular Fabio Massacci (University of Trento), Anna Perini and Angelo Susi (FBK-IRST), Nicola Guarino and his group (LOA-CNR). Thanks to Jaelson Castro (Universidade Federal de Pernambuco), who demonstrated interest in my work and provided valuable suggestions throughout my doctorate. Eric Yu (University of Toronto) and Oscar Pastor (Universidad Politecnica de Valencia) hosted me during visiting periods. Thanks for the opportunity to present and discuss my work, meet your research groups, and for the warm hospitality.

Thanks to all my colleagues in Trento, Valencia, and Toronto. I apologize for not listing your names. I think it would be unfair for the people I would certainly forget to mention. It was a pleasure to interact with you during my doctorate. I hope we will keep in touch. Thanks to all my friends: the nice time we spent together reminded me of the importance of social life.

Last but not least, thanks to my parents Flavia and Antonio, to my brother Luca, and to Francesca for their encouragements during my doctorate. You have been the greatest psychologists ever. The mood of a PhD student—*my* mood—is extremely variable. Thanks for standing me when I was in low mood, and thanks for your advices and unconditional love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems operate in dynamic environments, wherein expected and unexpected changes threaten their capability to meet their requirements. Developing self-adaptive software seems to be a promising way to cope with such volatility. Traditional approaches to self-adaptivity are conceived from a technical standpoint, and conceive adaptation as the adoption of a different technical configuration. However, many software systems do not operate in a purely technical environment. Conversely, they engage in social relationships with social actors (humans and organizations). This interplay of technical systems and social actors constitutes a Socio-Technical System (STS). In this chapter, we revisit software adaptation from a socio-technical perspective.

1.1 The age of socio-technical systems

There is no question that the software we *interact* with today is far more complex than that *used* just a few years ago. Think of a not-so-complex web application such as eBay. eBay enables you to conduct business: you browse the eBay.com website, select an item and place a bid, or you purchase immediately using the "Buy It Now" option. Alternatively, you can create auctions for your own items (that you do not necessarily own) and, as the auction finishes, ship the item to the winner through an express courier.

Importantly, modern software systems weave a network of business relations. eBay engages you in business transactions with the auctioneer or the bidders, payment processors, shippers, and the eBay corporation. Some people open personal shops on eBay, whereas some companies open virtual branches. Essentially, software acts as a means to create such relations on behalf of the social actor it represents.

The eBay website creates a number of business relations between you and the eBay corporation: you commit to deliver an item to the auction winner, eBay is entitled to suspend your account under certain circumstances, eBay commits to refund you if an item you bought is not delivered, etc. Compare this to the application you used less than two decades ago, when the Web was just born and desktop applications were mainstream.

Software is rapidly shifting from a computational setting where it works in isolation—or with few hard-coded interconnections with other technical systems—to social settings where it is part of a broader system composed of both technical and social subsystems. The term *Socio-Technical System* (STS) was coined in organizational theory to refer to a complex interplay of human actors—the traditional constituents of organizations—and technical systems [Eme59]. Software is a prominent example of technical system in an STS; humans and organizations exemplify social systems.

We encounter many examples of STSs in our daily lives. A smart-home that helps heart patients carry out everyday activities constitutes an STS. It includes a smart-home controller software, cameras, biomedical sensors, and other devices, human actors such as the patient itself, social workers, caregivers, doctors. The LinkedIn[1] social network defines an STS consisting of the LinkedIn website (technical system), professionals looking for a job, business scouts searching for interesting profiles, companies advertising open positions, and the LinkedIn corporation that aims at increasing revenues by offering professional relationships management. Likewise, a logistics department in a wholesale fruit company is an STS. A logistics management system supports purchases while warehouse personnel ensures fruit arrives to the warehouse just in time for delivery.

We rely here on the characterization of socio-technical system proposed by Ropohl [Rop99], which is founded upon general systems theory [VB73]. At the basis of his characterization is the notion of *action system*, a system that acts to transform a starting situation into a final one according to pre-set *goals*. Action relates not only to work (modifications in the environment), but also to communication with other systems. Ropohl asserts that a socio-technical system *is* an action system.

The principle of *excluded reductionism*, from general systems theory, states that systems are hierarchical. An STS can be understood in terms of internal subsystems that effectively execute the actions of the STS. These subsystems need informational coordination and communication to operate effectively. Communication and work are regulated by the goals of the STS and of its subsystems. In turn, each subsystem is an action system and can be hierarchically decomposed. Though not explicitly mentioned by Ropohl, this property supports a third kind of subsystem: organizations. Along with humans, organizations are the social components in an STS.

The principle of *equi-functionality* states that the function of a system may be produced by different structures. In an STS, actions can typically be performed by human subsystems as well as by technical subsystems. Only goal-setting cannot be performed by

[1]www.linkedin.com

technical subsystems. STSs are characterized by *socio-technical division of work*, the distribution of acting functions among humans and machines. This applies because technical systems are largely equi-functional with human abilities.

The discussion above suggests that an STS is an action system composed of both social and technical subsystems. Notice how the goal-orientation of STSs naturally leads to adaptation. Indeed, socio-technical systems should adapt to ensure their goals are met. Given their hierarchical structure, this results in different divisions of labour across the subsystems. To enact adaptation, the subsystems would typically consider the current division of labour and select the alternative that differs the least.

1.2 Software design in the age of STSs

The network of social relationships that arises and evolves in an STS has a deep impact on each participating subsystem. A single subsystem—either technical or social—cannot be merely understood in isolation, for this would ignore its relations with other subsystems. We are concerned here with a specific type of technical system, software. Specifically, our objective is to understand how *software design* needs to evolve to cope with the distinctive features of STSs, and to determine if and how well current design paradigms are suited.

Over the years, software evolved from monolithic structure to computational *distribution*. Pervasive [Sat01] and ubiquitous [Wei91] computing represent distribution brought to its extreme, where software becomes *transparent* to its users being well-camouflaged in the environment. Moreover, software is often *cross-organizational*: it enables different companies to make business together, as well as to build virtual organizations [Mow94]. For instance, service-oriented computing [PG03] lets each organization maintain its internal logic private and interact with others by exposing the interface of offered services.

Another relevant trend in software design is *adaptivity*. Adaptive (and self-adaptive) software emerged as a way to cope with the volatility of operational environments, which poses threats to the satisfaction of system goals. Software adaptation has been considered at different levels: some approaches emphasize the role of architecture in adaptation [OGT$^+$99, GS02]; some others provide programming frameworks to program adaptive behaviour [AHP94, AST09]; requirements-based approaches ensure that the system adapts to fulfil its purpose [FFvLP98, WMYM09]; service-oriented approaches analyse adaptation in terms of composition and recomposition [TP04, BDNGG07]; multi-agent systems literature provides algorithms for a single agent [Mye99, vdKdWW03] and environments that enable self-organization for a set of agents [MZ04, DNDM09].

Requirements Engineering has paid some attention to the social dimension during software design. This is particularly valid for goal-oriented requirements engineering [DvLF93, Yu96, vL01]. A notable example is *i**, an agent- and goal-oriented requirements modelling framework. *i** is a social modelling framework in the sense that it depicts an organization as a set of actors that depend on each other for achieving goals, executing tasks, and providing resources. However, the social relations that are identified at requirements-time are not necessarily preserved in the system to-be.

The challenge. The evolution of software design does not adequately take into account the social perspective that is so crucial for STSs. Although software design techniques do recognize the decomposition principles of STSs, social subsystems are typically not considered as first-class citizens. Consequently, overall STS designs are less than ideal. Our challenge is to *provide design techniques that lead to software that is operable in STSs*. In particular, this new class of software has to provide the following key features:

- **Take into account interaction with technical and social subsystems.** An STS consists of both technical and social subsystems. Software designed to operate in an STS should explicitly take into account such characteristic and be able to effectively interoperate with other subsystems. A smart-home controller should be able to interact with technical systems such as sensors and effectors, also to establish social relationships with social actors, such as patients, nurses, and catering services.

- **Operate in open systems.** Open systems [HdJ82, Hew86] are applications composed of multiple subsystems developed independently. They are founded on the *communication between subsystems*. As observed by Singh *et al.* [SCDM04], open systems involve *autonomous* and *heterogeneous* subsystems, and they are specified by the interaction among these subsystems. Many socio-technical systems are open systems. Consequently, software operating in an STS cannot control other participants—due to their autonomy—and cannot know how others reason—due to their heterogeneity and autonomy. The smart-home controller, for instance, cannot control the patient nor can it be sure of her intentions.

- **Deal with volatility via adaptation.** The environment where an STS operates is very dynamic. Therefore, participants—in our case, software—should be able to successfully cope with the threats of such volatility. To prevent failures, software should be able to self-adapt. Self-adaptation consists of the selection and enactment of a variant behaviour that better deals with the situation at hand. Different variant behaviours for software in an STS typically include different interactions with other subsystems. We consider two major types of volatility:

  - *The physical context is volatile.* Consider a smart-home that supports a patient in everyday activities. Patient's health varies, as well as his location in the house, temperature and humidity in the various rooms, availability of food in the fridge, network connectivity. Moreover, devices in the smart-home suffer of sudden disruptions and temporal unavailability;

– *The social context is volatile.* New actors (subsystems) can join and leave the system as they wish. Additionally, they add, remove, and modify services offered to others. Social actors in a smart-home STS vary over time. New social workers come into play, some others leave the STS, catering services offer different service levels and price, neighbours might be available or not. Notice that these actors might—intentionally or not—leave even when the patient relies on them. For instance, the nurse in charge of measuring the patient's blood pressure may have a car accident while reaching the smart-home.

1.3 Evolution and adaptation

The problem outlined in Section 1.2 is part of the broader challenge of software evolution. In this section we draw a parallel between natural (biological) evolution and software evolution. On the one side, this enables us to adapt concepts and mechanisms that operate successfully in nature. On the other side, this comparison helps us to determine the main differences between natural and software evolution.

Natural/biological evolution is the change in the inherited traits of a population of organisms through successive generations [Fut05]. This means that our descendants will be different from us in some trait. Software evolves as well: versioning might be considered as a basic evolutionary mechanism for software. New versions are released to ameliorate previous ones by adding, replacing, or dropping some features. However, evolution is not always ameliorative, neither in nature nor in software.

A key ingredient for evolution is variation, the existence of variants in a population. Variation is a result of a complex interplay of processes that introduce variation and others that remove it. The concepts of software variation and software variant refer to the existence of multiple instances of the same system class/family. Each instance differs from others in the features it delivers, i.e. in the way it meets requirements.

How is variation introduced in nature? *Mutation* introduces changes in a genomic sequence, and is caused by radiation, viruses, as well as errors that occur during meiosis or DNA replication. *Genetic recombination*—the basic mechanism underlying sexual reproduction—shuffles genes into new combinations which can result in new individuals exhibiting different traits. Less frequent—but still significant—mechanisms are the transfer of variation between species [JRL99] and the incorporation of genes through endosymbiosis [TAHM04].

Software developers artificially introduce variation in software systems. The current mechanisms to introduce variation are quite different from natural ones. Software product families are exploited to derive a number of variants with different characteristics; developers select which are the features to deploy. Variants arise when a project splits to different branches; this is the case, for instance, of the PostgreSQL and Ingres databases and of the many Linux distributions based on the same kernel.

In nature, there are two main mechanisms to produce evolution. *Natural selection* favours genes that aid the survival and reproduction of the population. *Genetic drift* is the random change in the frequency of alleles, and is caused by the random sampling of genes during reproduction. Natural selection can be easily found in software, and refers to the survival of software that better fits with the current environment—the software variant that best meets users' expectations. Genetic drift does not traditionally apply to software, since it arises from random recombinations.

The outcome of evolution are the changes that observers can see. We list the major outcomes in nature and show how they map to software:

– *adaptation* is the process whereby an organism becomes better suited to its habitat [May82]. Adaptation is noticeable by the modification of some trait in the exemplar: either a new feature is gained, or an ancestral feature is loss. Also software adaptation is characterized in terms of the features that are added, removed, or replaced. Software adaptation is performed to enable software to better fulfil its purpose in the current operational environment. For instance, the smart-home controller might be enriched with a feature to check patient's blood sugar level, if the doctor discovers the patient suffers from diabetes.

– *co-evolution* occurs when a biological entity changes in response to the change of a related entity. For instance, the production of tetrodotoxin in the rough-skinned newt produced the evolution of tetrodotoxin resistance in its predator, the common garter snake. Software co-evolution usually refers to the joint evolution of different software-related artefacts, e.g. implementation and design [DDVMW02]. Another type of co-evolution occurs in software suites (e.g. office suites) where the evolution of one application requires other applications to evolve to maintain interoperability.

– *speciation* makes a species diverge into two or more descendant species. In software, this is the case of a software project that is split into multiple variants, that differ for some feature from the original one. Differently from adaptation, speciation generates new entities whose identity is different from the original one.

– *extinction* is the disappearance of an entire species. In software terms, this corresponds to the retirement of a software system.

This thesis focuses on adaptation. More specifically, we consider self-adaptation, the capability of a software system to adapt without the intervention of external forces. We partially consider co-evolution: software in an STS interacts with other social and technical systems, therefore the evolution of any of these systems might require co-evolution by the considered software.

1.4 Research objectives

We outlined the research challenge tackled by this thesis—provide design techniques that lead to software operable in STSs—in Section 1.2, and we have shown how this relates to the broader theme of software evolution in Section 1.3. We specify now our challenge in terms of research objectives and questions.

Research Objective: *define a systematic process for designing self-adaptive software for socio-technical systems.*

We decompose the research objective into five specific research questions.

RQ1. What are software adaptation and variability?
As argued earlier, adaptation and variability are fundamental traits for STSs. Indeed, they enable software (and any other system) participating in an STS to cope with the volatility of their physical and social context. We propose to provide a precise account for them, as a prerequisite for tackling the following research questions.

RQ2. Which is an adequate conceptual model?
We employ a model-driven approach for the construction of self-adaptive software. One or more models are integral part of software and are used to diagnose failures, to identify new alternatives, and to keep track of the current configuration. We will address this research question by proposing a conceptual model that—based on the notions of adaptation and variability in RQ1—is effective for self-adaptive software in STSs. Our conceptual model will represent software requirements, explicitly support social relations, deal with variability in an efficient way, and provide traceability links from requirements to implementation.

RQ3. How can self-adaptive software be architecturally designed?
Software operates successfully only if its architecture is adequately structured. We will devise a general architecture for self-adaptive software based on several principles. First, requirements models should be an integral component of the architecture that keeps them alive at runtime. Second, the architecture should perform model-based diagnosis by checking monitored events against requirements models. Third, variability has to be explicitly considered: the architecture generates variants, selecting the most adequate, and enacts it

if needed. Fourth, the architecture should be designed for socio-technical systems. Thus, it derives existing social relations from interaction and exploits these relations during both system operation and adaptation. Finally, the architecture has to be applicable to several settings (i.e. should not be application specific).

RQ4. Which are effective and general adaptation algorithms?
Adaptation is the transition from the current configuration—variant—to another one that better satisfies system requirements. The effectiveness of self-adaptive software depends on the usage of sound adaptation policies and algorithms. First, adaptation should be performed only if needed and if the expected benefit from the new variant is higher than the effort required to adapt. Second, efficient algorithms should be devised to perform diagnosis and to plan and select new variants. Third, the algorithms should be general and rely on domain-independent criteria. Domain-specific criteria might be applied by tuning the algorithms. In a long-term view, a catalogue of adaptation policies should be defined to collect reusable adaptation strategies.

RQ5. How well does the approach perform when applied to realistic settings?
Any engineering approach needs empirical evidence to assess its applicability and performance. To evaluate our approach we shall conduct thorough experiments to verify whether it can be applied to realistic socio-technical systems and to assess its scalability. Specifically, we will verify if (i) the conceptual model is sufficiently expressive to represent requirements and social relations in STSs; (ii) the conceptual architecture can be adequately implemented and applied; and (iii) the proposed adaptation algorithms and policies are efficient.

1.5 Application areas

Our approach should be applicable to several types of socio-technical system. To such extent, the notion of adaptation has to be defined in terms of high-level and domain-independent abstractions. We describe three STSs we will use throughout this thesis to exemplify the introduced concepts.

Smart-home scenario. A smart-home is a home environment that supports and assists elderly or handicapped people (e.g. people with chronic heart problems) to live their lives. In smart-homes, an ambient intelligence technological infrastructure along with supervisory software are deployed in order to support and monitor inhabitants during their everyday activities around the clock. Typically, several devices—sensors and effectors—are deployed in the smart-home and function transparently to the inhabitants. Such devices are used to measure temperature, proximity and distance, force, pressure, and touch. RFID technology has also been recently exploited in smart-home settings, e.g. for authenticating people. As well, general-purpose devices, such as cameras, are employed for motion detection and trajectory recognition. Effectors actuate changes to the environment, e.g. servomotors for doors and windows, automatic 911 callers, light switches, displays, and remote controllers for multimedia devices such as TVs.

Suppose the smart-home, while monitoring the health of its inhabitant Bob, detects that he has a heart attack and has collapsed on the floor. The smart-home reacts to this event by requesting paramedic services. Since there is uncertainty on how quickly paramedics can be on the scene, the smart-home also requests assistance from a neighbour and proceeds to unlock the entrance door as the neighbour approaches. Alternatively, a police patrol may be dispatched in order to deliver first-aid services.

As an alternative scenario, suppose that Bob is to have breakfast within one hour after he wakes up, and that he needs to take his medicine just after breakfast. If Bob has not had breakfast and the hour is almost expired, the smart-home might send gentle reminders such as turning on a light positioned on the fridge to attract his attention. If such strategy doesn't work and the hour is expired, the smart-home might call a catering service or check whether a social worker can assist the patient in having his breakfast. □

Emergency response coordination. Emergency response is the phase of emergency management that includes mobilization of emergency services as well as responses to an emergency. Core actors involved in emergency response are fire brigades, police, and ambulances. Emergency response is a well-thought out activity for most countries. Emergency response plans are defined by municipalities, counties, regions, also at the national level. These plans assign responsibilities to different actors. Recent plans involve also the usage of an integrated information system to facilitate the collaborative efforts of participating actors. Emergency response unfolds in a very dynamic setting, and therefore adaptation is a required skill. For the supporting information system, this might mean detecting threats to the plans currently executed, also being able to identify opportunities to better deal with the emergency.

Suppose a fire-fighter squad requests additional water, for no fire hydrant is located in the fire area. If a traffic jam threats on-time arrival of the truck, the squad might request air delivery, if the weather is good enough for water tanker planes. If that is not the case, the squad might interact with another squad that is dealing with a low-priority emergency. In case the fire gets worse and toxic substances are burning, the squad might adapt its strategy and use chemicals instead of water, also contact a specific team that deals with hazardous substances. □

Mall Promotion Information System. Consider now a system that promotes products to customers inside shopping malls. It is assumed that both customers and sales staff are provided with PDAs to communicate and interact. The way the system supports the

customer depends on the specific context where it operates: the customer, the product, the sales staff and the shopping mall itself. A promotion process is initiated when a customer is within the mall building and is therefore able to accept a promotion offer. In choosing a promotion, the system has different alternatives: (i) cross-selling by suggesting a combination of products, if the customer is interested in a product for which a complementary product exists; (ii) offer a discount by showing a discount code, if the product is under promotion or is outdated; (iii) give free samples by calling a staff member, if free samples for that product exist.
To fulfil its requirements, the system has to continuously adapt. If the customer is interested in an item, and cross-selling is not working well (i.e. the customer is not interested in the cross-sold product), the system might either propose another cross-selling option or give a discount code. Similarly, if cross-selling is being offered but another customer has just checked out the last item of the cross-sold product, the system has to adapt and choose another option. □

1.6 Approach overview and contribution

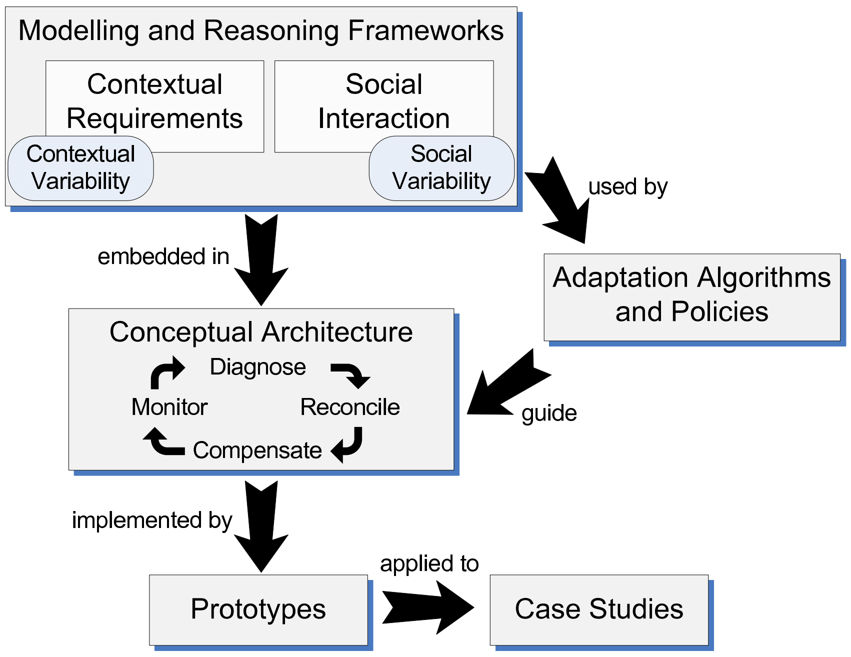


Figure 1.1: Overview of our approach to self-adaptive software

Our approach to self-adaptive software is outlined in Figure 1.1. We provide a comprehensive approach that goes from problem understanding and conceptualization to empirical evaluation via case studies. We overview now the building blocks of our approach, showing the tackled research questions and the contribution beyond the state of the art.

– **Modelling and Reasoning Frameworks.** We devise modelling primitives to represent contextual requirements and social interaction, respectively. Their shared baseline consists of Tropos goal models [BPG$^+$04]. Each language includes a graphical modelling notation as well as supporting automated reasoning techniques:

  – The *contextual requirements* framework associates contextual annotations to variation points in goal models. Each of these annotations describes when and where certain options (variants) in the goal model are applicable. Each contextual annotation is analysed to determine how its validity can be assessed at runtime via monitoring.

  – The *social interaction* framework is founded upon the notion of social commitment [Sin98]. Commitments represent social relations between heterogeneous and autonomous agents. They are publicly verifiable since they arise from interaction (the messages that are exchanged between agents). They capture the meaning of interaction, rather than describing interaction as a sequence of exchanged messages. Our framework integrates commitments with goal modelling. This enables to reason about how an agent can achieve its current goals, given a set of commitments that hold in the STS.

  Addressed research questions: RQ1 and RQ2.
  Contribution beyond state of the art: the modelling frameworks enable to explicitly deal with contextual and social variability at a high-level of abstraction. Our approach supports a different variability space than other goal-oriented approaches. Wang *et al.* [WM09] support variation points in goal trees. Souza *et al.* [SLRM10] extend Wang's work with control parameters. In our approach, the supported variability space consists of variation points, context, and the social interactions between systems (the enactment of dependencies via commitments).

– **Conceptual Architecture.** We propose a reference architecture that describes the basic components of self-adaptive software for STSs. Our architecture is model-driven, and relies on requirements models. To satisfy its requirements, the system can choose among alternative variants. The applicability of each variant depends on the current context, the social relations the software is involved in, and the possible relations it might establish. Our architecture is based on a Monitor-Diagnose-Reconcile-Compensate (MDRC) control loop. The system cyclically (i) monitors the surrounding environment (both the physical and the social context); (ii) diagnoses failures and under-performance by checking monitored data against models; (iii)

identifies possible variants to reconcile system behaviour with correct behaviour, as well as selects the best variant; and (iv) compensates the problem by enacting the selected variant.
Addressed research questions: RQ3.
Contribution beyond state of the art: differently from existing architectures for self-adaptive software (e.g. [OGT$^+$99, GS02, KM07]), ours is expressly suited for socio-technical systems. It preserves the autonomy and heterogeneity of subsystems, for it does not control other subsystems nor requires knowledge about their rationale. This is possible because social relations—commitments between subsystems—are decoupled from intentions—the goals of the supported system. If agent x intends to satisfy goal g through a commitment offered by y, then x need not know why y promises to deliver g or how y will bring about such goal.

– **Adaptation Algorithms and Policies.** We devise a set of adaptation algorithms that are used at runtime during the adaptation control loop:

  – We propose diagnosis algorithms that determine failures on the basis of monitored data. Failures occur when observed data deviate from data predicted by the models at-hand. Additionally, we consider proactive adaptation triggers to deal with opportunities the system has to improve its performance.

  – We define algorithms to enumerate possible variants to support current goals and to select the best variant. In particular, we present two algorithms for variant selection. The first one balances two factors: (i) it maximizes the contribution of the system to soft-goals, and (ii) it minimizes the difference from the current variant. The second one is a cost-based framework where goals are supported via capabilities and commitments; the best variant is that having minimal cost. Additionally, it can be customized by (i) specifying which events types trigger adaptation and (ii) enabling adaptation to a new behaviour only if significantly better than the current one.

  Addressed research questions: RQ4.
  Contribution beyond state of the art: most approaches to self-adaptive software pay little attention to the algorithms used in the adaptation control loop. In our approach, we specify algorithms for diagnosis—understanding what went wrong—and for reconciliation—identifying and selecting possible variants. We rely on widely applicable concepts—such as soft-goals and cost—to keep our algorithms general.

– **Prototypes and Case Studies.** We propose two prototype implementations of the conceptual architecture that highlight different features of our approach.

  - The first implementation focuses on contextual requirements—expressed as contextual goal models [ADG10]—and adapts in response to failures. This prototype selects the best variant by maximizing contribution to soft-goals and minimizing compensation cost. This implementation is applied to an Ambient Intelligence case study where a patient lives in an adaptive smart-home that monitors his health and supports him in everyday activities. The case study is adapted from the Serenity project[2].
  - The second implementation focuses on social interaction and variability. It enables the system to achieve its goals in a dynamic social environment where social relations vary. Such volatility consists of participants that join and leave the system as they wish, new commitments made by the system itself or by other systems, threats to and violations of existing commitments. This implementation exploits our cost-based adaptation framework, and lets system administrators specify adaptation policies. The implementation is applied to a simulated emergency response setting.

Addressed research questions: RQ5.
Contribution beyond state of the art: the implementations and their application to case studies allows us to identify bottlenecks and limitations of our approach. These implementations demonstrate the applicability of our conceptual architecture, and evaluate the effectiveness of the algorithms in practice.

1.7 Structure of the thesis

The thesis is structured as follows:

- Chapter 2 provides an extensive survey of the state of the art. First, we review modelling languages in the areas of requirements engineering and social interaction. These modelling languages represent the baseline for the models used in our approach. Then, we examine approaches to self-adaptive software. These approaches are classified into different categories to better contrast and position related work.

- Chapter 3 presents a study of the notion of variability and its role in self-adaptive software. We present two modelling frameworks that extend Tropos [BPG$^+$04] to deal with two facets of variability: contextual and social. For each framework, we describe how each factor—the physical context and social interaction—influences

[2]SERENITY (System Engineering for Security and Dependability) is an R&D project funded by the EU commission under the FP6: www.serenity-project.org.

system operation, we present the modelling framework, and emphasize how such framework deals with variability.

- Chapter 4 presents our conceptual architecture. It applies to multi-agent settings, where several subsystems (agents) interact within the scope of an overall socio-technical system. First, we present the underlying principles (the requirements) for the architecture. Second, we detail the logical view on the architecture via a component diagram. Finally, we show how it can be applied to an existing system.

- Chapter 5 introduces diagnosis and reconfiguration algorithms. The diagnosis algorithms allow for efficient detection of failures by checking monitored data against requirements models. While identifying new variants, reconfiguration algorithms consider not only qualities, but also to what extent each variant differs from the current one. We propose an adaptation framework for STSs with heavy social variability that enables the specification of adaptation policies.

- Chapter 6 describes two prototype implementations of our architecture; these prototypes implement the algorithms presented in Chapter 5. The first prototype focuses on contextual variability. It supports the entire adaptation cycle, from events monitoring to the actual variant enactment. The second prototype is specific for settings with social variability, and selects variants based on choosing commitments and capabilities to achieve current goals.

- Chapter 7 evaluates our approach on two case studies and presents scalability results. First, we model each case study using our frameworks for contextual and social variability. Second, we apply the prototypes to the case studies: we describe simulated scenarios where the prototypes perform adaptations. Third, we present scalability experiments for each prototype.

- Chapter 8 draws conclusions for this thesis, describes ongoing work, and presents future directions.

1.8 Published work

We list here published work related to this thesis. They are split into refereed (with subcategories for journal, conference, workshops, and others) and un-refereed (book chapters).

1.8.1 Refereed

International Journals

1. Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. *A Goal-based Framework for Contextual Requirements Modeling and Analysis.* Requirements Engineering, Vol. 15, Nr. 4, pp. 439–458, 2010.

2. Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. *Modeling and Analyzing Location-based Requirements: Goal-oriented Approach.* International Journal of Computer Science and Software Technology (IJCSST). Vol. 2, Nr. 2, pp. 89–95, 2009.

International Conferences

3. Fabiano Dalpiaz, Amit K. Chopra, Paolo Giorgini, John Mylopoulos. *Adaptation in Open Systems: Giving Interaction its Rightful Place.* In proceedings of the 29th International Conference on Conceptual Modeling (ER 2010), Springer LNCS 6412, pp. 31–45, 2010.

4. Amit K. Chopra, Fabiano Dalpiaz, Paolo Giorgini, John Mylopoulos. *Modeling and Reasoning about Service-Oriented Applications via Goals and Commitments.* In proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE'10), Springer LNCS 6051, pp. 113–128, 2010.

5. Amit K. Chopra, Fabiano Dalpiaz, Paolo Giorgini, John Mylopoulos. *Reasoning about Agents and Protocols via Goals and Commitments.* In proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), pp. 457–464, 2010.

6. Fabiano Dalpiaz, Paolo Giorgini, John Mylopoulos. *An Architecture for Requirements-driven Self-Reconfiguration.* In proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE '09), Springer LNCS 5565, pp. 246–260, 2009.

7. Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. *A Goal Modeling Framework for Self-Contextualizable Software.* In proceedings of the 14th International Conference on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD'09), Springer LNBIP 29, pp. 326–338, 2009.

8. Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. *Location-based Software Modeling and Analysis: Tropos-based Approach.* In proceedings of the 27th International

Conference on Conceptual Modeling (ER 2008), Springer LNCS 5231, pp. 169–182, 2008.

9. Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. *Location-based Variability for Mobile Information Systems.* In proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE 08), Springer LNCS 5074, pp. 575-578, 2008.

10. Raian Ali, Fabiano Dalpiaz and Paolo Giorgini. *Modeling and Analyzing Variability for Mobile Information Systems.* In proceedings of the International Conference on Computational Science and Its Applications (ICCSA 2008), Springer LNCS 5073, pp. 291–306, 2008.

Workshops

11. Raian Ali, Amit K. Chopra, Fabiano Dalpiaz, Paolo Giorgini, John Mylopoulos, and Vitor E. Silva Souza. *The Evolution of Tropos: Contexts, Commitments and Adaptivity.* In the proceedings of the 4th International *i** Workshop, co-located with CAiSE 10. CEUR-WS Vol-586, pp. 15–19, 2010.

12. Fabiano Dalpiaz, Raian Ali, Yudistira Asnar, Volha Bryl, Paolo Giorgini. *Applying Tropos to Socio-Technical System Design and Runtime Configuration.* In the Proceedings of the 9th WOA workshop, From Objects to Agents (Dagli Oggetti Agli Agenti), ISBN 978-88-6122, 2008.

Other refereed publications

13. Fabiano Dalpiaz, Paolo Giorgini, John Mylopoulos. *Software Self-Reconfiguration: a BDI-based approach (Extended Abstract).* In proceedings of the 8th International Conference on Autonomous Agents and Multiagent System (AAMAS 2009), pp. 1159–1160, 2009.

14. Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. *Goal-based Self-Contextualization.* In the Forum of the 21st International Conference on Advanced Information Systems (CAiSE 09 - Forum). CEUR-WS Vol-453, pp. 37–42, 2009.

1.8.2 Un-refereed

15. Amit K. Chopra, John Mylopoulos, Fabiano Dalpiaz, Paolo Giorgini, Munindar P. Singh. *Requirements as Goals and Commitments too.* Book chapter in: Intentional Perspectives on Information Systems Engineering, Springer, 2010.

Chapter 2

State of the art

The state of the art in the area of self-adaptive software is broad and includes very diverse approaches belonging to different research sub-areas, including conceptual models, software architectures, service-orientation, requirements engineering, algorithms, agent reasoning, and self-organization. We overview these sub-areas and assess their adequacy in supporting self-adaptation for socio-technical systems. Section 2.1 reviews literature that defines our research baseline, covering Requirements Engineering, Social Interaction, and Software Variability. Section 2.2 provides a comprehensive survey on approaches to self-adaptive software.

2.1 Baseline models for self-adaptive software

Our approach to self-adaptive software is model-driven: models are used at runtime to detect adaptation needs and to identify alternative behaviours. Thus, our baseline inevitably consists of modelling concepts and techniques. According to the big picture we presented in Chapter 1, we need models that represent:

– The purpose of the system, captured by requirements models. We review goal-oriented modelling languages in Section 2.1.1.

– The social relations the system is currently or might potentially be engaged in. Social relations are the glue that ties together different participants in an STS, and they are established and modified by the interaction between participants. We review modelling frameworks for social interaction in Section 2.1.2.

– Software variability, the possibility to achieve requirements through different alternatives. A model for software variability should represent variants in a structured way and relate variants to the requirements of the system. We discuss software variability models in Section 2.1.3.

2.1.1 Modelling requirements via goal models

Goal-Oriented Requirements Engineering (GORE) has emerged as a prominent approach within Requirements Engineering (RE). According to GORE, stakeholder needs are goals —desired states-of-affairs—and should be elicited, modelled and analysed accordingly. There is a large amount of work in the area, which encompasses all phases of RE [vL00]: domain analysis, elicitation, negotiation and agreement, specification, specification analysis, documentation, and evolution.

A common thread in all GORE research is that goals and their refinements are captured by goal models using a formal or semi-formal notation. Goal models serve as a communication artefact among requirements engineering and stakeholders, but also as abstract specifications of the system-to-be. We review here the most influential goal modelling approaches and show their main strengths, specific features, and limitations.

The NFR framework [MCN92, CNYM00] has been devised to model and analyse Non-Functional Requirements (NFRs). Apart from the proposed technical framework, this work is a paradigmatic shift in requirements engineering, for it motivates the need for and emphasizes the importance of non-functional requirements. Indeed, the authors treat NFRs as selection criteria for design decisions during software development. NFRs can be formally treated either from a product-oriented or a process-oriented perspective. The first perspective means to develop formal definitions for NFRs. The second perspective—addressed in the NFR framework—proposes their usage for justifying decisions during the software development process. An orthogonal dimension is qualitative versus quantitative treatment. The authors observe that obtaining quantitative measurements is too hard a task, and select thus qualitative treatment of NFRs. The NFR framework consists of five major components: (i) a set of goals that represent NFRs, design decisions, and arguments in support or against other goals; (ii) a set of link types that correlate goals to other goals; (iii) a set of methods for refining goals into others; (iv) a set of correlation rules for inferring potential interactions among goals; and (v) a labelling procedure that decides to what extent a NFR is satisficed. They use the term "satisficed", instead of "achieved" or "satisfied", to suggest that there is no concept of full satisfaction for a NFR, for they represent qualities that are satisfied within acceptable limits, rather than absolutely. The framework supports three types of goals: (i) non-functional requirements of the system to-be, (ii) satisficing goals that represent design decisions, and (iii) arguments that provide evidence or counter-evidence for other goals. Design proceeds by refining one or more times each goal using different types of links: AND/OR decomposition, four types of contribution, equivalence, undetermined. The model where these concepts are combined is called Soft-goal Interdependency Graph (SIG). Developers choose alternative combinations of leaf-level soft-goals and use a label propagation algorithm to verify how well the different alternatives satisfice the high-level non-functional requirements.

KAOS (Knowledge Acquisition in autOmated Specification) [DvLF93] is a goal-oriented methodology heavily focused on the formalization of requirements. The authors' claim is that representing requirements—conceptualized as goals—formally enables automated reasoning. They suggest KAOS as a supporting methodology for the requirements acquisition phase, where designers aim to understand which are the requirements of the system to-be. Central to their framework is the concept of goal model, a graph where each node captures an abstraction (goal, action, agent, entity, event), and whose edges represent semantic links between such abstractions. The conceptual model of KAOS enables to express both functional and non-functional requirements. KAOS includes also an acquisition process to construct requirements models and an automated support tool. The conceptual meta-model of KAOS consist of three levels: the meta-level represents domain-independent abstractions (e.g. agent, action, relationship); the domain level describes concepts specific to the application domain (e.g. resource management, telephone network); the instance level refers to specific instances of domain-level concepts. According to the KAOS world-view, objects are things of interest which can be referenced in requirements; objects are specialized into entities, relationships, events, and agents. Agents decide on their own behaviour. The KAOS requirements acquisition process consists of eight steps: it starts with the acquisition of an initial goal structure and the identification of concerned objects, and it ends with the assignment of actions to agents. Roughly, the six intermediate steps refine the initial structure and identify responsibilities.

The Goal-Based Requirements Analysis Method (GBRAM) [AP98] is a method to identify, elaborate, refine, and organize goals for requirements specification. GBRAM relies on a conceptual model consisting of (i) goals, high-level objectives of the business; (ii) requirements, specifications of how a goal can be fulfilled by a system; (iii) operationalization, the refinement of a goal so that its sub-goals have an operational definition; (iv) achievement and maintenance goals whose condition should be met once or kept true, respectively; (v) agents, the entities that seek to achieve goals within an organization; (vi) constraints that place conditions on the achievement of goals; (vii) goal decomposition, the process of subdividing goals into logical subgroups; (viii) scenarios, behavioural descriptions of a system and its environment that specify a certain situation; (ix) goal obstacles, behaviours or goals that prevent the achievement of a given goal. The GBRAM process consists of two phases: goal analysis and goal refinement (or evolution). Goal analysis concerns the identification of goals starting from process descriptions, their organization, and their classification. Goal refinement is about how goals change from the time they are first identified to the time they are operationalized. Goal refinement is very important since stakeholders change their minds very often over time. GBRAM does not provide any graphical notation to represent and structure goals; rather, it makes use of textual tables in so-called goal schemas.

Rolland *et al.* [RSA98] propose a method that integrates goal modelling with scenario authoring. Their approach supports goal identification through scenarios. They focus the discovery of goals around the concept of Requirement Chunk (RC), a pair ⟨Goal, Scenario⟩. They define a bidirectional goal-scenario coupling: goals help in scenario discovery, and scenarios help in goal discovery. Consequently, requirements elicitation consists of two steps: scenario authoring and goal discovery. Another key feature of their work is the distinction between the refinement relationship and the AND/OR relationships among goals. Requirements chunks are organized in a requirement chunk model, which keeps track of all RCs and their refinements. Refinement is used to go from high-level fuzzy goals to lower-level concrete goals, whereas AND/OR relations create a hierarchy for each concrete goal. The structure of both goals and scenarios is specified through a conceptual model that helps defining these artefacts on the basis of natural language statements. For example, a goal has at least one verb and at least one parameter, where parameters can be target, direction, way, and beneficiary. This approach also provides guidelines to assist the designer during requirements identification.

*i** [Yu96] is an agent-oriented modelling framework usable for requirements engineering, business process re-engineering, and organizational modelling. The novelty of *i** is that, unlike other approaches, the notion of agent is a first-class citizen. In KAOS, for instance, operationalizable goals are assigned to different agents on the basis of a responsibility principle. In *i**, goals exist only if associated to a certain agent; in other words, agents want to achieve goals, either using their own capability or depending on another agent. When using *i** for requirements engineering, two phases are to be carried out. During the early requirements phase, *i** is used to model the stakeholders of the system, their objectives, and their relationships. *i** models developed during early requirements help understanding why a new system is needed. During the late requirements phase, *i** models are used to propose possible ways to construct the new system and the new organizational processes. The alternatives can be evaluated and compared based on how well they meet functional and non-functional requirements. Though agent-oriented, the *i** ontology includes a broader abstraction, that of actor. Actors can be agents (concrete actors with specific capabilities), roles (abstract actors embodying expectations and responsibilities), or positions (socially recognized roles). Actors are intentional: they have goals they aim to achieve. Goals are decomposed through AND/OR decomposition and means-end relations into other goals, tasks, or resources. *i** supports two types of goals: hard and soft. Soft-goals are goals for whose satisfaction level there is no clear cut criteria, as in the NFR framework [MCN92]. Actors depend on each other for the fulfilment of goals, the execution of tasks, and the provision of resources. Intentional elements—goals, tasks, and resources—may contribute positively or negatively to other intentional elements. The

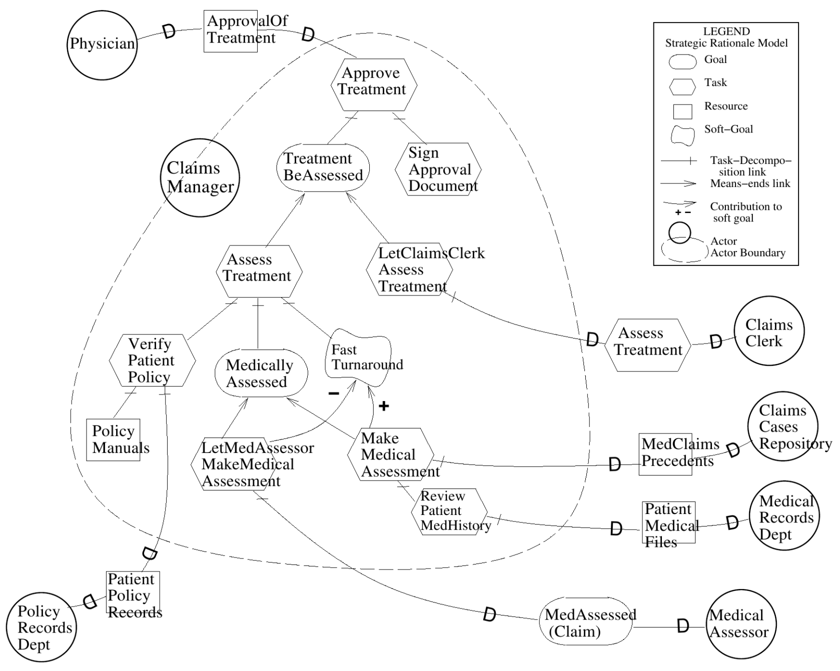


Figure 2.1: A strategic rationale model in *i** [Yu96]

*i** framework exploits these abstractions in two models: Strategic Dependency (SD) and Strategic Rationale (SR). An SD model is a network of dependency relationships among actors, and captures the intentionality of the processes in the organization and what is important to its participants. SR models are exploited to describe the rationale behind the processes in systems and organizations. In SR models, the rationale of every actor is described in terms of intentional elements: goals, soft-goals, tasks, and resources, and the relationships among them. Figure 2.1 shows how a strategic rationale model looks like.

Tropos [BPG^{+}04] is an agent-oriented software engineering (AOSE) methodology that relies on the *i** meta-model. Unlike *i**, Tropos is a methodology that exploits mentalistic notions (e.g. goals and tasks) from early requirements to actual implementation. Tropos consists of five phases: early requirements analysis, late requirements analysis, architectural design, detailed design, and implementation. From the point of view of the underlying conceptual model, Tropos starts from *i** and adds some restrictions. For example, decompositions are allowed only between homogeneous elements (goals can be decomposed only to sub-goals, tasks to subtasks, resources to sub-resources); plans can be means-end decomposed only to plans or resources, and not to other goals.

2.1.2 Modelling social interaction

Social interaction arises naturally, both in our physical environment and, increasingly, in our virtual (software) world. Social interaction occurs whenever two or more actors interact (typically, for some purpose). Examples of everyday social interaction are visiting a doctor, buying a car from a car dealer, exchanging information about summer holidays. Social interactions often involve technology—in particular, software systems. Think of your payments via credit card, where you interact with a software system that processes the payment, your credit card company, the item seller. Social interaction creates social relationships between the interacting participants. There is a large amount of research about social interaction modelling; the most prominent approaches are in agents and multi-agent systems, conceptual modelling, software and service engineering.

Odell *et al.* propose AUML (Agent Unified Modeling Language) [OPB01], an extension of UML that supports, among other features, agent interaction protocols. Their proposal is intended to increase adoption of the agent technology in industry through the definition of a standard modelling framework that extends UML. Their notion of agency requires only (i) the capability to initiate action without external invocation and (ii) the ability to refuse or modify an external request. In AUML, an agent interaction protocol (AIP) describes a communication pattern as an allowed sequence of messages between agents and the constraints on the content of those messages. AUML adopts a layered approach to protocols, where interaction protocols can be specified in more details using a combination of diagrams. Level 1 represents the overall protocol at a high-level of abstraction, in order to facilitate reuse of the protocol. AUML uses packages to express nested protocols and templates, parametrized model elements whose parameters are bound at model time. Level 2 represents interaction between agents using the dynamic models of UML: sequence diagrams, collaboration diagrams, activity diagrams, and statecharts. The combined usage of all these diagrams allows to specify interaction between agents in a process-like fashion. Level 3 describes internal agent processing, i.e. how agents are structured to carry out an interaction protocol. This is done through activity diagrams and statecharts.

Some approaches to agent interaction are based on the speech acts theory by Searle [Sea70]. More specifically, these approaches rely on indirect speech acts [Sea75], that require interaction parties to understand the motives behind utterances. For example, a sentence such as “Can you pass the salt?”, though syntactically an interrogative that expresses a question, is actually a request message. There are two standard languages developed by the Agents community: the Knowledge Query and Manipulation Language (KQML) [FFMM94] and the FIPA Agent Communication Language (FIPA-ACL) [FIP97]. We present here only FIPA-ACL as an example of agent communication language based on indirect speech acts. The interested reader may find more on the similarities and differences between these two languages in the work by Labrou *et al.* [LFP99]. FIPA-ACL describes how messages exchanged by agents should be structured. Being based on speech acts, a performative field is needed to represent the communicative act of the message. Examples of communicative acts are accept, agree, cancel, call for proposals, confirm, inform, failure, request, and query. Three fields are used to represent participants in the conversation: sender, receiver, and reply-to. The content is the core of the message, which is interpreted by the receiver agents according to the communicative act, the encoding, the ontology, and the protocol. FIPA-ACL contains some fields to represent control of conversation: the protocol to which the message belongs to, a conversation ID that uniquely represents different conversations, reply-with and in-reply-to to deal with conversations where multiples dialogues occur simultaneously, reply-by to indicate response deadlines.

In his "Rethinking the principles" paper on agent communication [Sin98], Singh observes that agent communication languages do not let heterogeneous agents communicate, and that approaches that rely on mental agency are unrealistic. If agents are autonomous and heterogeneous, the sender cannot make any hypothesis about the receiver's rationale. Singh shows that an agent communication language has to emphasize social agency. Communication is inherently public and depends on the agent's social context. The long-term challenge he poses is the definition of agent communication languages that (i) are based on formal semantics; (ii) support high autonomy; (iii) enable high heterogeneity; (iv) support open dialects. Singh calls for a radical shift from previous agent communication languages, that make many assumptions concerning the structuring of the single agents—violating their heterogeneity—and have very little flexibility.

Kumar *et al.* [KHC02] propose a landmark-based approach to agent interaction, where landmarks are those states of affairs that must be brought about during a goal-directed execution of a protocol. In such approach, interaction protocols are joint action expressions that can be derived from partially ordered landmarks. Interaction protocols are then executed directly by joint intention interpreters. Unlike traditional approaches to agent communication, this work asserts that the correctness of a protocol is defined in terms of the states to be achieved, not of the state transitions (i.e. the messages). The formalization of protocols in terms of partially ordered states (the landmarks) allows for the automated verification of a protocol. Also, interaction protocols composition is supported to derive composed protocols from simpler ones. Despite of its execution flexibility, this approach—being based on joint action theory—works only for cooperative agents, and does not take into account open settings (such as socio-technical systems) where agents are autonomous and heterogeneous. Indeed, the hypothesis about the existence of a joint intention violates the principles stated by Singh [Sin98]; joint intentions are not necessary—nor typical—for social interaction.

Cheong and Winikoff [CW06] propose Hermes, a goal-oriented approach to agent interaction. Hermes is to be used by engineers that want to define interaction protocols for multi-agent systems, and includes (i) a methodology to design goal-based interactions; (ii) failure handling mechanisms; and (iii) a process to map design artefacts to an executable implementation. They observe that message-centric approaches are not suitable for flexible and robust interaction, being rooted in low-level abstractions. Such inflexibility—typically expressed by the sequencing of messages to be exchanged—restricts the autonomy of agents and does not allow the exploitation of shortcuts (reduced versions of a protocol). In Hermes, an interaction protocol is a set of goals to be achieved and some temporal constraints about these goals. The Hermes methodology supports the design of an interaction protocol in six steps, starting from the identification of roles and interaction goals and ending with the definition of messages to exchange. Hermes supports action failure and interaction goal failure. An action failure occurs when an action does not achieve its interaction goal. An interaction goal failure occurs if an interaction goal cannot be achieved. The main limitation of this approach is to suppose that goals exist at the level of an overall multi-agent system: this assumption holds when designing an entire MAS, but does not hold in case of autonomous and heterogeneous agents, which can join and leave as they wish.

To overcome the limitations of existing agent communication languages, Singh proposes the concept of social commitment [Sin99] as the interaction abstraction for multi-agent systems. In his work, commitments are described and exploited in a multi-agent architecture named spheres of commitment. His work on commitments relies on several assumptions: agents are autonomous (and constrained only by social commitments to avoid chaos), social commitments cannot be reduced to internal commitments, social commitments are revocable through appropriate operations, commitments exist only in a social context, commitments are different from the way agents use them, commitments are a social abstraction—not a mental one. Formally, a commitment is a four-place relation involving a proposition p and three agents x, y, and G. A commitment $c = \mathsf{C}(x, y, G, p)$ denotes a commitment from x to y in the context of G for the proposition p. x is called debtor, y is the creditor, G is the context group, and p is the discharge condition of commitment c. The semantics is that the debtor x commits to the creditor y to bring about p in the context G. The social context G includes norms and conventions that apply in the environment where the commitment c is instantiated. Commitments can be manipulated via specific operations performed by the involved agents: a debtor creates a commitment, a debtor discharges a commitment by satisfying the discharge condition, a debtor cancels its commitment—implying a violation, a creditor can release a debtor from a commitment, delegation transfers the role of debtor to another agent, assignment transfers the role of creditor to another agent. A commitment is conditional if the debtor commits to the discharge condition only if a certain precondition holds. In subsequent work, Singh and colleagues consider mainly conditional commitments—which are particularly adequate to represent business relations—as a five-place relation $c = \mathsf{CC}(x, y, G, p, q)$, in which p is called antecedent, q is called consequent, and the debtor x commits to y for q if p holds.

Commitments have been used by Desai *et al.* [DMCS05] to define flexible interaction protocols for business processes. In their approach, a business protocol is an abstract, modular, and public specification of an interaction among different partner roles. They contrast protocols defined in terms of commitments to traditional message-flow based protocols, and show that their approach is more flexible and robust. They also show how composite protocols can be defined. Their framework is formalized in π-calculus in order to allow inference reasoning about (in)correctness, compatibility, equivalence, and flexibility. Telang and Singh [TS09] apply the concept of commitment to the Tropos methodology in order to enable to specify cross-organizational business processes. The motivation of this work is similar to that of Desai *et al.* [DMCS05], but the approach is different. Here, a business process consists of a set of agents having goals and tasks, and a set of roles related via dependencies and commitments. The two abstractions related to interaction—dependencies and commitments—are used at different steps of the methodology. First, goals and goal dependencies between roles are identified. Then, goals and goal dependencies are refined into tasks and task dependencies between roles. Finally, task dependencies are further refined into conditional commitments. The authors explain the importance of verifying agent interaction, since agents autonomy does not guarantee that their interaction will follow the protocol defined as commitments between roles.

2.1.3 Software variability modelling

Software variability refers to “the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context” [SvGB05]. Software variability is an essential property for adaptive systems, for software adaptation does not exist in absence of variability. We review here the most influential approaches to software variability modelling.

Kang *et al.* propose Feature-Oriented Domain Analysis (FODA) [KCH$^+$90], the first domain analysis technique to apply the concept of feature modelling to domain engineering. Feature models are not only beneficial to domain analysis, but also they promote software reuse. The proposed method allows for discovering and representing commonalities among related software systems. The core concept used to define commonality is that of feature: “a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems”. Feature models are not the only component of FODA, but they its distinctive trait. Figure 2.2 shows a feature model for a car. A car has two

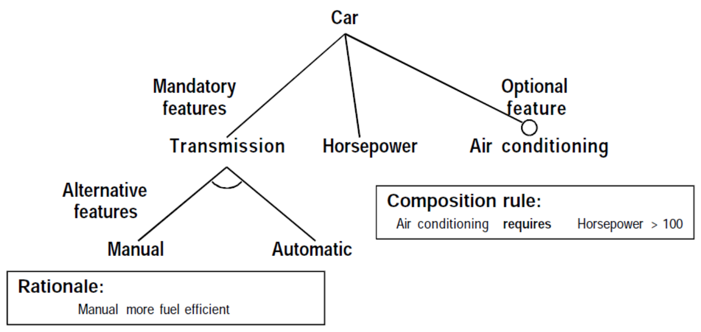


Figure 2.2: A feature model using the FODA notation [KCH+90]

mandatory features—that every car instance has—transmission and horsepower. Cars have optional features—that a car instance may have or not—such as air conditioning. Alternative features are those that vary from one instance to another; transmission can be manual or automatic. Optional and alternative features represent variation points, i.e. constructs where variability is introduced. The feature model in Figure 2.2 contains two additional element: a composition rule saying that air conditioning requires horsepower to be bigger than 100; and a rationale saying that manual transmission is more fuel efficient than automatic transmission. These two elements allow for more expressiveness and help designers to choose among variants.

The same authors extend FODA and propose FORM [KKL+98], which focuses on the development of reusable architectures and components. While FODA focuses on requirements only, FORM supports the entire software design phase. FORM prescribes how the feature model is used to develop domain architectures and components for reuse. FORM is based on abstraction principles, and is composed of four layers. Every layer exploits feature models; higher-level features are refined in lower-levels. The four supported layers are: capability, operating environment, domain technology, and implementation technique. FORM enables to transform a feature model to a reference architecture. Such transformation takes into account non-functional features, which influence the structure of the architecture.

Czarnecki *et al.* [CHE05] introduce Cardinality-Based Feature Modeling (CBFM), an extension of FODA that enhances the expressiveness of feature models. Apart from the contribution given by the extension itself, their work provides a formal account of feature modelling with a semantic interpretation of feature models. They exploit a notion of feature that is more generic than that by Kang and colleagues: "a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family". Such definition allows for features defined with respect to any stakeholder, not just users. Features can be of two kinds: solitary features and grouped features. Grouped features appear in feature groups. Cardinality-based feature models express cardinalities as intervals over integers. Their approach also proposes the concept of staged configuration, the process of specifying a product family member in stages, where each stage eliminates some configuration choices. The formalization of the cardinality-based feature model is in terms of context-free grammars.

Von der Maßen and Lichter [vdML02] study how UML use case diagrams can express variability. They first define the requirements for a variability modelling language: representation of common and variable parts, distinction between types of variability, representation of dependencies between variable parts, support to model evolution, and provision of good tangibility and expressiveness. Their baseline is the UML use case diagram, which is extended by adding the concepts of variation point, alternative, and option. Halmans and Pohl [HP03] propose a different way to express variability via use case diagrams. First, they show how the "include" and "extend" links can be used as a primitive means to represent variability. Second, they observe the inflexibility of such solution and propose an alternative approach based on UML stereotypes. Specifically, they introduce the concept of variation point to explicitly take into account variants.

Asikainen *et al.* propose Koalish [ASM04], a modelling language to express configurable software product families supporting the automatic configuration of product instances. The language extends Koala [vOvdLKM02], a component model and architecture description language, by adding explicit variation modelling mechanisms. Unlike most approaches in variability modelling, Koalish does not define variation points, while it allows for selecting the number and type of parts of components, and writing constraints. A Koalish model represents the properties of a software product family and describes component and interface types that can occur in configurations. A configuration is a set of instances of these types; a configuration is valid if it satisfies the constraints in the Koalish model.

Van der Hoek [vdH04] observes how existing approaches to variability modelling support variability in a specific phase of the software life cycle, and proposes an approach to supporting any-time variability. The key novelty of his approach is to exploit a product line architecture as the organizing abstraction during the entire life cycle of software. His approach relies on four components: a representation to capture variability, a tool to specify variability, a generic tool to resolve variability, and specialized tools that at different points in the life cycle apply the results of variability resolution. For the purpose of our work, we focus on the language he chooses to express variability. He suggests the use of xADL 2.0, an extensible architecture description language built as a set of extensible XML schemas. xADL is composed of nine XML schemas. Van der Hoek exploits the six lower-level models to represent the structure and the functionalities of a specific architecture. Fundamental to express variability are the Options, Variants, and Boolean Guard schemas. Together, these schemas allow for modelling variability in space through explicit variation points in the architecture. The Options schema supports the definition of optional architectural elements; the Variants schema enables to define variant elements that can be configured to be one of a set of alternative types; the Boolean Guard schema allows to define the preconditions for the Options and Variants schemas.

Gomaa and Webber propose the Variation Point Model (VPM) [GW04]. The authors outline four approaches to modelling variability: parametrization, information hiding, inheritance, and variation points. The variation point model belongs to the last category. Parametrization means changing some core attribute of the application, e.g. varying a greeting depending on the device display. Information hiding relies on the implementation of different version of a same component sharing the same interface, e.g. developing a multi-language ATM based on different language components. Inheritance is also based on components but relies on specialization (e.g. by extending a generic class); thus, the components need not have the same interface. VPM is based on an explicit representation of variation points, that are contained in reusable components. VPM uses UML models to describe variability and consists of four points of view. The requirements point of view shows the variation point in terms of the variations of the common core. The component variation point view is a composite view that shows all the variation points that belong to a particular component. The static point view exploits an UML class diagram to show structural variability. The dynamic variation point view shows how to make a variant known to the other components.

Liaskos *et al.* explore the use of goal models for variability representation and analysis [LLY+06]. The core idea is to explore OR-decompositions to represent alternative ways of satisfying goals. They associate a set of concerns to high-level goals and introduce alternative refinements of the goal to address each of these concerns. The concerns they define are (i) agentive, the agent whose activities will bring about the state of affairs related to the goal; (ii) dative, the agent affected by the generic activity implied by the goal; (iii) objective, the object affected by the generic activity implied by the goal; (iv) factitive, the object that results from the activity; (v) process, the instrument, means and manner to perform the activity; (vi) locational, the spatial location where the activity is performed; (vii) temporal, the duration or frequency of the activity; (viii) conditional, the conditions under which the goal can be fulfilled; and (ix) extent, alternative degrees by which the generic activity can be performed. They show how these concerns can be applied to perform concern-driven decomposition of a goal model, so that the variants in a goal model keep track of the concerns they address.

2.2 Approaches to adaptive software

In this section we present our literature survey on adaptive software. We organize the state of the art according to the categories listed in Table 2.1. Approach that fall into more categories are described in the most representative category.

Category	**Section**	**Description**
Conceptual models	2.2.1	Provide conceptual foundations to conceive and understand adaptation
Programming frameworks	2.2.2	Enable the development of adaptive software by extending programming languages
Software architectures	2.2.3	Define architectural styles or are frameworks to build adaptive software
Service-oriented	2.2.4	Conceive adaptation in a setting composed of multiple interacting (web) services
Requirements engineering	2.2.5	Guarantee that the system fulfils its requirements
Algorithms and policies	2.2.6	Define criteria about how to adapt and enable designers to specify adaptation policies
Agent reasoning and planning	2.2.7	Characterize adaptation from the perspective of an intelligent agent
Self-organization	2.2.8	Adaptation in a multi-agent system results from the individual behaviour of the agents

Table 2.1: Categories of approaches to adaptive software

2.2.1 Conceptual models of adaptation

Some approaches address software adaptation from a conceptual perspective, rather than through a technical framework. The main questions they tackle are: what is adaptation?; why and how should software adapt?; how does adaptive software differ from non-adaptive one?; and when should an adaptation process start?

Zhang and Cheng [ZC06] introduce a formal model for the behaviour of self-adaptive software. Their purpose is to structure software so that it works correctly both during and after adaptation. In their approach, they separate adaptation models from non-adaptation models. By decoupling these two models, it becomes easier to understand and formally verify adaptive features, for they are isolated from non-adaptive behaviour. They formalize adaptive software as a program whose state space can be separated into a number of disjoint regions (programs), each exhibiting a different steady-state behaviour and operating in a different domain. Adaptation models guide the transition from a source program to a target one. Programs are formalized via petri nets as finite state machines. Their conceptualization of adaptation is general and can be applied to architectural models, UML diagrams, or formal requirements modelling languages.

Salehie *et al.* [SLAT09] propose a conceptual model for adaptation changes based on activity theory. They conceive adaptive software as a closed-loop system that aims at self-adjusting under different situations at runtime. Specifically, they focus on two different facets of adaptation: *how* adaptation is carried out and partly *where* it applies, i.e. which is the object to change. They rely on activity theory to define an adaptation change as "an activity which sets to modify an artefact of software to attain specific objectives". Their approach refines the concepts of activity and objective hierarchically. Activity is broken down into actions and operations, whereas objective is split into goals and conditions. The two hierarchies are then related by a formal framework for adaptation changes. Though applied to a single example concerning fine-grained adaptation changes, their model is applicable to any type of adaptation, at different levels of granularity.

Andersson *et al.* [AdLMW09] classify the modelling dimensions for self-adaptive software systems. Each dimension constitutes a particular facet of the system that is relevant to adaptation. These facets refer to four macro-categories: system goals, causes of self-adaptation, mechanisms to achieve self-adaptivity, and effects of self-adaptation. They characterize system goals along different dimensions: evolution (static/dynamic), flexibility (mandatory/optional), duration, multiplicity, and interdependency. Change triggers should describe the source of change (internal/external), type of change, frequency, and anticipation (can change be foreseen?). Adaptation mechanisms can be classified according to their type (parametric/structural), autonomy (self/human-assisted), organization (centralized/decentralized), scope (local/global), duration, timeliness, and trigger type (event/time). Effects are modelled by considering criticality, predictability, overhead, and resilience. Their taxonomy helps adaptation frameworks designers to assess their approach and to compare it to different approaches. Also, their work suggests some important dimensions to consider when proposing new approaches.

Taxonomies of software change and evolution consider not only runtime adaptation, but also other types of software change. Mens *et al.* [MBZR03, BMZ$^+$05] characterize how, when, what and where, of software changes. To address these questions, they define a taxonomy of characteristics of change mechanisms and the factors that influence these mechanisms. These characteristics are related to software change either as characterizing factor, influencing factor, or both. An orthogonal classification of these characteristics is based on the question they address. Temporal properties, that answer the "when" question, are time of change (static, load-time, dynamic), change history, change frequency, and anticipation. The object of change (where) should consider the artefact that evolves, granularity, impact, and change propagation. System properties (what) are availability, activeness, openness, and safety. Change support (how) should explain degree of automation, degree of formality, and change type. This taxonomy can be applied to classify approaches to runtime software adaptation. However, being more generic than [AdLMW09], it results less informative.

McKinley *et al.* [MSKC04] make the case for compositional adaptation. They contrast this form of adaptation to parametric adaptation, and define it as the exchange of algorithmic or structural system components with others that improve a program's fit to its current environment. Compositional adaptation is supported by three main technologies: separation of concerns, computational reflection, and component-based design. At runtime, it is enacted by middleware that supports the components replacement. Separation of concerns, understood as aspect-oriented programming [KLM$^+$97], is very important to dynamic recomposition because many adaptations refer to a cross-cutting concern, such as quality of service. Computational reflection is fundamental to let the program reason about its current state, identify the need of adapting, and find a better configuration. Component-based design is suggested as the enabler for software components replacement.

Inverardi *et al.* [IPT09] propose a theory for software adaptation based on the preservation of a set of invariants during adaptation. Theirs is an assume-guarantee framework that allows to efficiently define the conditions under which adaptation should be performed while guaranteeing the desired invariants. Being independent of the level of abstraction, their theory can be applied at different levels of abstraction (from architecture to code). The main feature of their approach is the usage of assume-guarantee reasoning, which enables the verification of a component-based system through the verification of the single components. The main limitation is that they presume adaptive software can be neatly split into independent components, which is seldom the case in real systems.

2.2.2 Programming frameworks

Programming languages and framework provide developers with programming abstractions that support and facilitate the development of self-adaptive software systems. These programming frameworks propose a clear shift from traditional programming methods and styles. Indeed, they provide developers with specific languages and data structures to create self-adaptive applications.

Agnew *et al.* [AHP94] propose Clipper, a C++ extension that facilitates the development of reconfiguration plans responding to reconfiguration events. Clipper is applicable to distributed systems where multiple processes execute and are interconnected by communication channels. The approach consists of three steps: (i) creation of the reconfiguration module, (ii) application creation, and (iii) application execution. A core concept is that of application configuration, a collection of C++ structures. Clipper enables programmers to describe mappings between configurations as operations on structures. A reconfiguration plan is defined by associating each mapping with the conditions and events that trigger reconfiguration. The plans are then compiled into a so-called catalyst module which executes along with its the main application. The catalyst module recognizes triggering events and performs reconfiguration plans.

Asadollahi *et al.* [AST09] propose StarMX, a framework to develop self-managing Java-based systems. StarMX is intended for programming closed loops and supports several adaptation mechanisms, among which action policies. Like Clipper, adaptation (management) logic is decoupled from application logic. The architecture of StarMX consist of two main elements: the execution engine and a set of services. StarMX exploits the Java Management Extensions (JMX) as enabling technology to manage and monitor resources, and requires a policy language to define adaptation policies. The default implementation of StarMX uses the CIM-Simplified Policy Language [DMT09]. The execution engine automates self-management: a number of processes are deployed on it, each representing an adaptation building block. StarMX provides makes services available through an API: lookup of new resources, proxy generation, activation mechanism, caching, memory scoping (a repository mechanism), data gathering, and logging. These features make StarMX a comprehensive and generic framework applicable to several domains and customizable via specific algorithms. However, StarMX does not embed ready-to-use adaptation mechanisms.

Liu *et al.* [LPH04] propose Accord, a component-based programming framework to build autonomic self-managed applications. Accord is thought for pervasive grid environments, where a large number of computational devices is interconnected. This type of environment is characterized by heterogeneous components, dynamism, and uncertainty. The third factor is due to dynamism, failures, and incomplete knowledge. To cope with these factors, the Accord programming framework relies on four concepts: (i) the application context, which is a common semantic basis for the application; (ii) the autonomic components, self-contained modular software units of composition with specified interfaces and explicit context dependencies; (iii) the rules and mechanisms for the dynamic composition of autonomic components; and (iv) an agent infrastructure to support rule enforcement to realize self-managing and dynamic composition behaviours. In particular, The agent infrastructure used by Accord is the AutoMate middleware [PLL$^+$03]. Adaptation is performed by so-called composition agents, which replace components with new ones. Accord works well for computationally distributed grids, where components are passive structures that can be activated and deactivated by the middleware. Conversely, Accord does not support open settings where every component is autonomous and is hence not controllable.

Bigus *et al.* [BSP$^+$02] propose ABLE (Agent Building and Learning Environment), a toolkit for the development of multi-agent autonomic systems. ABLE consists of a lightweight Java agent framework, a comprehensive JavaBeans library of intelligent software components, a set of development and test tools, and an agent platform. ABLE relies on learning techniques, among which machine learning algorithms and inference engines. ABLE has been applied to server administration, where an ABLE-based multi-agent system monitors the server parameters, sends notification to system administrators, and performs maintenance operations (e.g. killing processes). ABLE is a complete development framework and provides a number of features for the development of autonomic systems. ABLE is best suited for closed systems, whereas it does not natively support socio-technical systems, where the subsystems need to interact.

Sadjadi *et al.* [SMCS04] propose TRAP/J, a software tool that enables developers to transparently add adaptive behaviour to existing Java applications. TRAP/J relies on two mechanisms to add adaptation: computational (behavioural) reflection and aspect-oriented programming. At compile time, developers select the classes that need to be self-adaptive. TRAP/J then generates specific aspects and reflective classes associated with the selected classes, producing an adapt-ready program. At runtime, new behaviour can be added to adaptable classes through specific interfaces in the wrapper and meta-level classes (i.e. the reflective classes). The same research group also proposed a variant of TRAP/J based on C++ (TRAP/C++ [FCSM05]), which uses a compile-time meta-object protocol instead of an aspect weaver.

2.2.3 Software architectures

There are many proposals of software architectures for adaptive software. These approaches describe how to design adaptive software, and which are the core logical components. Since software architectures are a higher-level abstraction than programming frameworks, these approaches are more flexible than those presented in Section 2.2.2. Some approaches are implemented architectures that can be applied, while others are architectural styles that define the basic components of adaptive software.

In their seminal work, Oreizy *et al.* [OGT$^+$99, OMT98] introduce the concept of self-adaptive software. This class of systems performs adaptation according to criteria specified at development time, which include triggering conditions for adaptation, open/closed adaptation, degree of autonomy. According to Oreizy and colleagues, the building units for self-adaptive software are software components and connectors. This leads to adaptations that modify the system architecture by replacing components. Their approach relies on four main features: (i) an explicit architectural model is available at runtime and describes the interconnections between components and connectors; (ii) runtime change is described on the architectural model; (iii) runtime change is governed via constraints to be preserved, i.e. some adaptations are admitted whereas others are forbidden; and (iv) adaptation is enabled by a reusable runtime architecture infrastructure that supports the previous three features. This approach has been very influential and most of the subsequent approaches rely on the principles they suggest. Their ICSE 1998 paper received the most influential paper award at ICSE 2008, where they proposed a revisited version of their paper [OMT08]. In this revision, they briefly review the state of the art in the area and survey architectural styles for software adaptation, which they consider the big challenge in the next years.

A relevant early work, which focuses only on monitoring and diagnosis, is that by Savor and Seviora [SS97]. They introduce software supervision, an approach to automatic detection of software failures. They are among the first ones to suggest that monitoring and diagnosis should be performed by a separated component, whose purpose is to observe inputs and outputs of a target system and to detect software failures. The key principle behind this work is to match observed behaviour and expected behaviour—through a finite-state based formalism—to identify failures. In their work they propose algorithms and mechanisms to address computational complexity by increasing latency.

Brun *et al.* [BMSG$^+$09] analyse the design of self-adaptive software through feedback loops. Feedback loops are an integral component of self-adaptive software, and define how the system reasons to exhibit runtime adaptive behaviour. Typically, feedback loops are based on four activities: collect, analyse, decide, and act. Sensors collect data from the system and its surrounding context; these data are then analysed to diagnose symptoms of failures or under-performance; planning is then performed to decide how to act on the executing system and on the context through actuators. The authors argue that feedback loops should be a first-class entity for self-adaptive software, which should be made explicit and expose the self-adaptive properties to designers. Consequently, they call for a shift in the development of feedback loops on the basis of systematic approaches used in control theory. Software architectures for adaptive software should therefore highlight the key aspects of feedback loops: their structure, data flow, tolerance policies, sampling rates, stability, and so on.

In a similar way, Müller *et al.* [MPS08] highlight the importance of feedback loops and exemplify their use in Ultra-Large Scale (ULS) systems. They observe that, by their nature, ULS systems lack of central control and do not have complete specifications. They argue ULSs should ensure that the system operates correctly, and they reckon self-adaptive architectures as the way to tackle this challenge. First, they position adaptivity (consequently, feedback loops) as a solution space property. Second, they sketch a reference feedback control loop for adaptive software. Finally, they observe that such control loop is compatible with several existing implicit feedback loops used in technical architectures.

Karsai *et al.* [KLS$^+$01] propose an approach to self-adaptive software based on supervisory control. They take this concept from control theory. Using terminology from that field, one can build self-adaptive software systems using a “Ground-Level” (GL) layer that includes baseline processing, then use a “Supervisory-Level” (SL) layer responsible for the adaptation and reconfiguration. The GL layer focuses on baseline functionality and performance, whereas the SL layer is concerned with optimization, robustness, and flexibility. They represent system configuration in terms of a tree-based hierarchical representation of components. In order to deal with the variability explosion problem, they suggest the usage of symbolic representation techniques (e.g. ordered binary decision diagrams), so that efficient decision procedures can be used.

The Rainbow project [GS02, GCH$^+$04] is an architectural approach for self-adaptive software based on externalized adaptation. This term refers to the capability of the system to keep alive, at runtime and external to the application, one or more models for identifying and resolving problems. Externalized adaptation is a response to the observation that most adaptation mechanisms are embedded in the application itself, at the code level, and therefore their reuse is very difficult and unlikely. Also, externalized adaptation is more generic than embedded mechanisms and allows for detecting soft system anomalies, such as gradual degradation of performance. Rainbow relies on architectural models of the application expressed using an Architectural Description Language (ADL). ADLs represent system architecture as a graph, where nodes are components, while arcs are connectors and represent the pathways of interaction between components. There has been a considerable amount of research concerning Rainbow. An interesting extension supports multiple objectives [CGS06]. The authors propose a language to express high-level stakeholder objectives and use utility theory techniques to choose the adaptation that best suits the objectives. A support tool has been developed to engineer adaptive systems [CGS09]. Another extension adds proactive behaviour, so that the architecture can anticipate failures and prevent their occurrence [CPGS09].

Kramer and Magee introduce self-managed systems [KM07]. They assert that the architectural level is the most suitable, in terms of abstraction and generality, to support self-management. With the term self-managed software they refer to a system whose architecture is one in which components automatically configure their interaction so that the overall purpose of the system is met. The objective of their work is to minimize human intervention during software design and maintenance. They propose a three-layer reference architecture—based on the model proposed by Gat in Artificial

Intelligence [Gat98]—shown in Figure 2.3. The topmost layer, goal management, is the

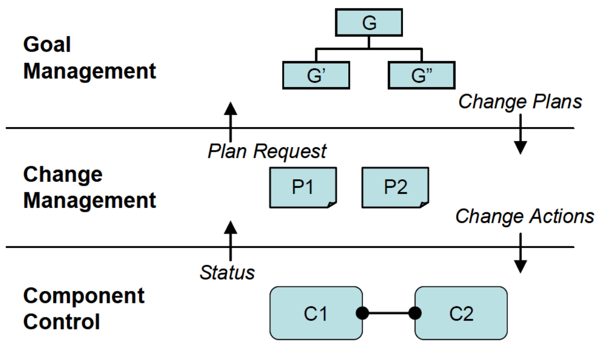


Figure 2.3: Self-managed systems three-layer reference architecture [KM07]

deliberation layer. Based on the current goals and state, this layer computes a plan to achieve these goals. The middle layer, change management, reacts to changes in state reported from the lower levels and reactively executes the plans the goal management layer provides. The bottom layer, component control, consists of sensors, actuators and control loops, and includes self-tuning algorithms, event and status reporting, and operations to support system modification (adding, removing, and interconnecting components). In subsequent work, Sykes *et al.* [SHMK08] describe the three layers in more detail. Their goal management layer accommodates the generation of reactive plans—that support a non-deterministically changing environment—by prescribing an action towards a given goal for each state from which that goal is reachable. Their change management layer uses a simplified component model based on Darwin [MDEK95], where a component has a set of ports, each requiring or providing a single interface. Their domain-specific component layer includes Java components based on the Backbone language [MKM06].

Autonomic computing [Hor01, KC03] is an IBM initiative aimed at the development of self-managing systems to overcome the growing complexity of traditional system management. Here, self-management is the capability of a system to manage itself given high-level objectives set by system administrators. Self-management includes four properties: (i) self-configuration, i.e. systems that autonomously adjust their configuration parameters according to high-level policies; (ii) self-optimization, i.e. continuous seek for opportunities to improve performance and efficiency; (iii) self-healing, i.e. autonomic detection, diagnosis, and repair of software and hardware problems; and (iv) self-protection, i.e. defence against malicious attacks or cascading failures. Autonomic computing systems consist of multiple interacting *autonomic elements*. Each autonomic element includes a managed element and an autonomic control loop implemented by an *autonomic manager*. The loop is founded on a Monitor-Analyse-Plan-Execute (MAPE) cycle. A successful prototype implementation of an autonomic computing system is Unity [TCW+04, CSWW04]. In addition to autonomic elements, Unity has several auxiliary components: (i) an application environment manager takes care of the operational environment (obtaining resources, communicating with other management elements); (ii) a resource arbiter decides about who should use a certain pool of resources; (iii) a registry enables elements to locate other elements to interact with; (iv) a policy repository enables system administrators to change policies; and (v) sentinel elements monitor other elements. Unity uses utility functions to determine resource allocation in order to maximize the overall system utility. Autonomic elements are able to compose with other elements to attain a certain goal (goal-driven self-assembly).

Floch *et al.* [FHS+06] propose MADAM, an architectural approach to develop adaptive software for mobile computing. Unlike other approaches, MADAM explicitly deals with context changes that threaten software operation. Their approach starts by monitoring context factors that can affect the application under consideration, such as battery, computing resources, and network quality of service. Users of the mobile application express their preferences. Then, the quality provided by the monitored context is compared to user preferred quality and, if needed, an adaptation middleware adapts the mobile application. To identify alternative operational modes, MADAM makes use of architectural models and computes possible application variants. Being expressly thought for mobile devices, MADAM relies on adaptation mechanisms that are computable on mobile devices. They use extended goal policies expressed as utility functions, and let the system reason about the actions needed to implement those policies.

CASA (Contract-based Adaptive Software Architecture) [MG05] is another approach that deals with contextual variability via self-adaptation. CASA differentiates between changes in the contextual information (user's location, identify of nearby persons) and changes in resource availability (bandwidth, battery power, connectivity). The authors argue that, in order to adequately deal with changes in the environment, both dynamic change support in application code and adaptation of lower-level services are needed. The CASA Runtime System (CRS) hosts a set of adaptive applications. The CRS monitors changes in the execution environment on behalf of these applications, and is responsible for adapting them whenever required. Every time the CRS detects a change in the environment, it evaluates the application contracts of the running applications to check if these changes trigger some adaptation policies. If adaptation is required, the CRS can effect it through various mechanisms: lower-level services are changed, aspects are weaved or unweaved, application attributes are modified, components are recomposed.

Vogel and Giese [VG10] observe that most architectural approaches to adaptation rely on architectural models that are as complex as the core system architecture itself. To overcome such complexity, they propose a model-driven approach that provides multiple architectural runtime models at different levels of abstraction. Moreover, they deal with different concerns. This way, their approach facilitates the development of adaptation managers. A source model is associated to a managed system to represent its architecture. Then, a model transformation engine is used to derive more target models, each representing a specific facet of adaptation. For instance, one target model could deal with self-optimization, another one with self-protection. Monitored data result in a first set of changes in the source model, then these changes are propagated to the target models via model transformations. When adaptation is needed, parameter adaptation and structural adaptation can be performed by modifying the specific target models, rather than changing the source model.

Cetina *et al.* propose an architecture for autonomic computing based on variability models [CGFP09]. Though applied to a smart-home case study, the approach is generic. They represent variability via feature models; these models are used at design-time to configure the system, and at runtime to determine adaptive behaviour. Their architecture supports (i) self-configuration, e.g. when new devices are added to the system; (ii) self-healing, e.g. when a device is removed or fails; and (iii) self-adaptation, e.g. when user needs change. Changes in the context are monitored and checked against variability models to detect problems. Then, a model-based reconfiguration engine defines a reconfiguration plan which is enacted to the actual system.

2.2.4 Service-oriented approaches

Organizations are rapidly evolving from stable and monolithic structures to dynamic and federated ones. It is more and more common to encounter virtual organizations where business is performed through delegation of work and subcontracting. Software is evolving according to the same trend. The most prominent computational paradigm to support these new organizational structures is service-orientation. Software services expose their functions through a public interface, that is exploited by other systems to identify and exploit these functionalities. Due to the volatility of organizational structures, service-oriented applications have to be flexible and adaptive. For instance, if a company X relies on a service provided by another company Y, and Y goes bankrupt, X has to find a different provider for an equivalent service, or its operations might be at risk. A comprehensive review on adaptive service-oriented applications is offered by Di Nitto *et al.* [DNGM+08]. They outline the evolution of organizations and computational paradigms, review current approaches for service-based applications, and highlight the need for novel service-based approaches that apply to modern settings.

Much research on service-orientation examines how services can be composed to tackle problems that a single service alone could not address. Two major approaches have emerged: service orchestration and service choreography [Pel03]. They describe different aspects of composing services thereby relating business processes from composite (web) services. These terms derive from music. In orchestration, services are composed by a player (the director) that directs their composition. In choreography, every service is a player, and services get composed as a result of their interaction, without the intervention of any director. The web-services community has defined several languages to support these operations. In particular, BPEL4WS [ACD$^+$03] is the standard for service orchestration, whereas WS-CDL [KBR$^+$04] is that for service orchestration. Most of the following approaches are based on composition and orchestration.

Some approaches enable automated composition of web services via automated planners. Among them, Traverso and Pistore [TP04] propose a planning technique that applies to web services described in OWL-S process models. Their approach can deal effectively with nondeterminism, partial observability, and complex goals. They synthesize plans that encode compositions using common programming constructs such as conditionals and iterations. They exploit the MBP [BCP$^+$01] planner, which relies on planning-as-model-checking. The inputs are OWL-S process models and a composition goal, while the output is an executable process expressed in BPEL4WS.

Lazovik *et al.* [LAP06] propose a planning architecture based on the interleaving of planning and execution. Their framework starts from an user request and tries to fulfil it against a business process expressed, e.g., in BPEL. The framework returns failure if the request is unsatisfiable in the given business process under the current runtime circumstances. Satisfiability is verified using a planner, thus unsatisfiability means that no plan exists. As a plan is chosen, the executor enacts the plan by invoking web services. User requests are expressed using the Xml Service Request Language (XSRL). This architecture provides adaptive behaviour by interleaving planning and execution. These two activities are iterated until the goal (the user request) is met. Though designed to respond to user requests, this framework can be easily extended to perform autonomic service composition and execution.

Li *et al.* [LSQC05] propose an approach to tackle the runtime reconfiguration of a service-based system. Their approach addresses a very specific kind of change: geometrical change. The logical application structure may remain fixed, while the mapping of the logical structure to physical hardware nodes—the geometry—changes. The configuration of a service system is composed of a set of machines, a set of services, and information concerning where these systems are deployed. Also, on the basis of a set of metrics, they define (i) a metric satisfaction function that returns true if a configuration satisfies a metric, and (ii) a configuration satisfaction function—which returns true if all active metrics are true. Their approach is based on a MAPE loop. Reconfigurations are initiated either reactively—in response to an SLA violation—or proactively—by resource over-consumption or under-utilization. Ill configurations are analysed to identify which are the services that require reconfiguration (which consists of service migration, in this work). Then, a planning algorithm is executed to identify the best reconfiguration, which is later enacted. The limitation of this approach is to presume that services in a service-based system can be controlled. Such assumption is invalid for STSs.

Denaro *et al.* [DPTS06] propose a way to develop self-adaptive service-oriented architectures. They observe that the task of service systems designers is hard due to the lack of information concerning the interaction protocols of dynamically discovered web services. This absence might cause unexpected runtime failures. Their approach enables clients to adapt their behaviour to alternative web services that provide the same functionality through a different protocol. They rely on an infrastructure that traces successful interactions of web services and approximates interaction protocols from these traces. This infrastructure, named Interaction Protocol Service Extension (IPSE), is deployed within a Service-Oriented Architecture. IPSE exploits efficient state of the art algorithms to derive interaction protocols from successful interaction.

Koning *et al.* propose VxBPEL [KSSA09], an extension of BPEL to capture variability in web-services based systems. VxBPEL supports four types of variability: replacement of a service by one with the same interface, replacement by one with a different interface, changing the service invocation parameters, and changing the composition of the system. They augment BPEL with `VariationPoint` and `Variant` tags to explicitly represent variability. They propose a prototype interpreter that understands VxBPEL. When it encounters a variation point, the prototype chooses the variant that is currently better performing. Performance is computed on top of either monitored data or QoS metrics.

Siljee *et al.* propose DySOA (Dynamic Service-Oriented Architecture) [SBNH05], an architecture that adds self-adaptation capabilities to service-centric applications. The purpose of DySOA is to maintain a certain QoS in the service system. Several events trigger adaptations, such as unreliable third-party services, user changes, and network irregularities. DySOA assists the service system in maintaining its QoS, which is known only at runtime depending on negotiation for service-level agreements. DySOA runs concurrently with the application system, monitors QoS parameters, and enacts reconfigurations when QoS is not met. Reconfiguration is based on an explicit variation model that documents variation points and allows for the selection of an alternative variant. Each variant is characterized by a realization, the instructions to realize and bind the variant.

Baresi *et al.* [BDNGG07] propose a framework for the deployment of adaptable web service compositions. The novelty of this approach is its publication infrastructure that integrates existing heterogeneous repositories and makes them cooperate for service discovery. Also, the behaviour of these repositories is adjusted in response to changes and unforeseen events. The main components of this approach are: (i) the DIstributed REgistry (DIRE) to which existing registries can subscribe; (ii) the Service Composition ExecutioN Environment (SCENE) that supports the execution of service compositions—using an extension of BPEL—and enables runtime recomposition; and (iii) the Dynamic monitoring (Dynamo), which provides runtime monitoring of BPEL-like processes. The overall infrastructure enables the engineering of service-centric systems capable of dynamic service (re)composition.

Dorn and Dustdar [DD10] address adaptation in service systems from a different perspective. They observe that adaptation cannot be conceived from the perspective of a single service alone, but it should take into account factors that emerge from the overall interaction in the service ensemble. They contrast the service ensemble—which comprises humans, software services, and the service infrastructure—to the system, which includes only software elements. They introduce a MAPE-K cycle that turns an autonomous system into an evolving system; such cycle (depicted in Figure 2.4) is based on ensemble-specific adaptation functions. The key feature of their approach is interaction monitoring, measuring the relations between different human and technical actors via a distance measurement. Distance decreases when two actors perform several shared actions and when these interactions re-occur. By continuously monitoring distance, system requirements can be adjusted. An example of requirement might be storage capability.

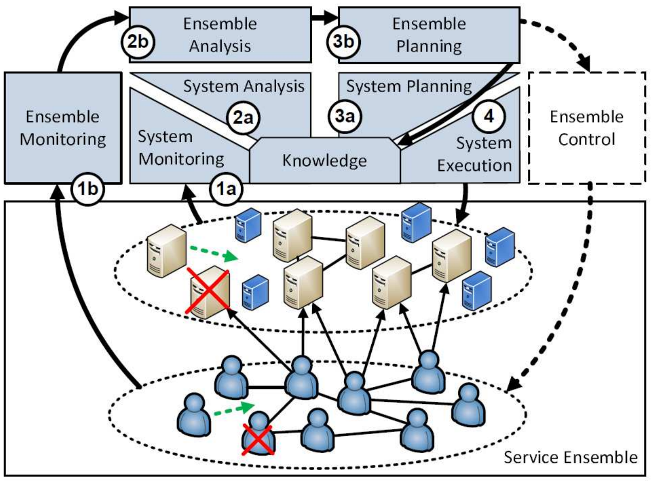


Figure 2.4: MAPE-K autonomic control loop with functions to adapt service ensembles [DD10]

Dustdar *et al.* propose a roadmap for self-aware sustainable service systems [DDL$^+$10]. They claim that the complexity of self-adaptive systems has become unmanageable, therefore there is a clear need for sustainable approaches. The major challenges for adaptation in service systems are (i) limited awareness on resource utilization and long-term effects; (ii) lack of understanding interdependencies between social and technical entities; (iii) local information in heterogeneous large-scale environments; (iv) dynamic and decentralized evolution of requirements, interests, and topology; and (v) no central authority. To tackle these challenges, they propose to make systems more self-aware in terms of: (i) event awareness, so that events trigger ECA rules; (ii) situation awareness, the ability to perceive the status of an entity by aggregating relevant events; (iii) adaptability-awareness, the capability of a service to detect the adaptability of other entities and to open its own interface of adaptation; (iv) goal-awareness, keeping track of the overall objectives of the system; (v) future awareness, i.e. understanding a resource's life-cycle describing long-term utilization by the system and resource provisioning by the environment. We agree with them on the importance of self-awareness; also, most of their observations inspire our work. However, unlike them, we also consider systems where services are not cooperative.

Cavallaro *et al.* [CDNFP10] propose a model for the design of self-adaptive service compositions based on the concept of service tiles. The key idea is to let designers create a service-oriented system by building an assembly of component services to achieve a certain goal. Their approach automatically computes an assembly from the specification of a subset of the whole system, a set of constraints, and the goals of the application. Once the application has been deployed according to the tiles model, it is capable of adapting by replacing services if context changes or services fail. With tiles, a designer first abstracts the structure of the application process; then, she builds a workflow of the application and defines which external services should be invoked. Each service is associated to a string that represents the goal that service fulfils. At runtime, the self-adaptive system can replace a service with another that provides the same goal.

2.2.5 Requirements engineering approaches

There is no question that software design should guarantee the fulfilment of software requirements. We review here approaches to self-adaptive software based on requirements models. These approaches state that adaptation should occur to prevent requirements failure and/or to optimize requirements fulfilment. Requirements-based approaches differ in the used requirements models and in the algorithms to identify failures and to select a new variant.

Some approaches perform requirements diagnosis only and leave adaptation to designers. Cohen *et al.* [CFNF97] propose one of the earliest approaches to automatic requirements monitoring. The same research group had previously motivated the need for approaches to requirements monitoring [FF95]. They claim that monitoring is practicable only if (i) requirements are expressed in a flexible and convenient way; (ii) requirements expressions are automatically compiled into efficient runtime code; (iii) monitoring is applicable to black-box systems; (iv) monitoring works in an incremental fashion. They support these requirements through an approach where requirements are expressed in FLEA (Formal Language for Expressing Assumptions). FLEA models are then compiled into a monitor for a target system, so that notifications about violated requirements (assumptions) are sent to administrators. More recently, Robinson [Rob06] proposes ReqMon, a requirements monitoring framework expressly thought for enterprise systems. He observes that, in modern organizations, requirements monitoring is a necessary activity. The modus operandi of ReqMon is comparable to the early work by Cohen and colleagues; however, the framework by Robinson is more comprehensive. Indeed, ReqMon consists of a language to define requirements, an assisting methodology, and requirements monitoring tools. ReqMon is based on the goal-oriented KAOS language [DvLF93], which is formally represented and allows for defining temporal expressions. On the basis of the KAOS-specified requirements, and with the support of automated reasoning techniques, ReqMon derives obstacles and defines the monitors to deploy in the system. Finally, Robinson suggests ways to implement these monitors as SQL monitors or ECA rules.

Robinson and Purao [RP09] propose SerMon, a requirements monitoring framework for systems composed of several interacting subsystems. Web services are a notable example of this kind of system. Their approach extends ReqMon to support: (i) commitment- and message-based specifications of interaction; (ii) inherited agent properties; and (iii) event acquisition from the Common-Base-Event enterprise monitoring framework. SerMon exploits rule-based monitors based on OCL extended with temporal-message logic (OCL_{TM} [Rob08]). Commitments are formally represented as invariants in OCL_{TM}, therefore they become additional requirements for the committed agent.

Bencomo *et al.* explore the notion requirements reflection [BWS$^+$10], which has been introduced by Finkelstein [Fin08]. Such concept specializes computational reflection, the ability of a program to observe and possibly modify its behaviour at runtime. They suggest to consider computational reflection at the requirements level, i.e. requirements are available as runtime objects. Requirements reflection provides systems with the ability to reason about, understand, explain and modify requirements at runtime. In traditional approaches, even when requirements monitoring functions are integrated within the system, high-level system requirements are manually refined into low-level runtime artefacts. Differently, requirements reflection reifies requirements as runtime entities. They are in favour of using goals to represent requirements at runtime for introspection and adaptation. Goals should be synchronized with the architecture, and this challenge can be addressed only through a middleware that propagates changes in requirements to the architecture. Requirements reflection should necessarily deal with uncertainty, for self-adaptive systems operate in a volatile environment where unexpected changes occur.

The first approach to requirements-based adaptation is that devised by Feather *et al.* [FFvLP98]. Their approach copes with system behaviour deviations from requirements specified as KAOS goal models. They introduce an architecture (and a development process) to reconcile requirements with behaviour. Their monitoring infrastructure is based on the FLEA system [FF95]. Alternative system designs are represented either as system parameters or as alternative goal refinement trees. To reduce the gap between requirements and runtime behaviour their approach relies on three phases: (i) at specification time, event sequences to be monitored are generated from requirements specifications; (ii) at design time, they build an architecture consisting of multiple cooperating agents and in which alternative designs are explicitly represented; and (iii) at runtime, the system is observed by a monitor and, if a violation is identified, a shift is made to an alternative design. Their approach is illustrated in Figure 2.5. KAOS requirements models are defined

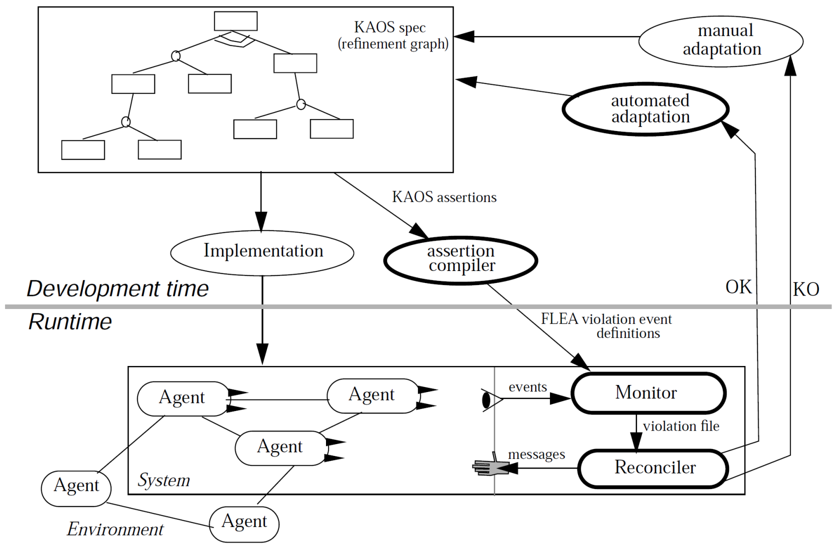


Figure 2.5: Requirements-based adaptation [FFvLP98]

at development time; then, requirements are implemented via a multi-agent system and assertions are compiled to FLEA violation event definitions. At runtime, the multi-agent system executes and provides input (events) to the monitor. If violations are identified, the reconciler sends messages to the multi-agent system to perform adaptation. If needed, the KAOS specification models are updated, either automatically or manually.

Wang *et al.* propose another approach to self-adaptive software based on $i*$ goal models. The first part of their work concerns requirements monitoring and diagnosis [WMYM07, WMYM09]: they exploit a single $i*$ goal model to express system requirements, and annotate goals and tasks with pre- and post-conditions. In their framework, a failure occurs if (i) a post-condition is met while the respective pre-condition does not hold, or (ii) an event representing a precondition occurs and at the next time-step the postcondition does not hold. They support diagnosis of failures by transforming goal models to satisfiability problems in conjunctive normal form, which are then fed into a SAT solver that performs the actual diagnosis. Their approach applies to legacy systems and requires the system to be instrumented with monitors so that it can provide monitoring data to the architecture. Technically, instrumentation exploits aspect-oriented programming: they use the AspectJ weaver to insert software probes into the monitored program without modifications to the source code. They provide algorithms that enable to tune the granularity of monitoring, so that a trade-off can be defined between monitoring overhead and accuracy. In [WMYM09] they show how the monitoring trade-off becomes relevant when the approach is applied to multi-layer monitoring, e.g. in service-oriented architectures. This framework has been extended to support self-repair [WM09] for high-variability software. The resulting architecture is an autonomic one based on monitoring, diagnosis, reconfiguration, and execution components. In their work, a configuration is a set of tasks from a goal model which, if executed successfully in some order, lead to the satisfaction of the root goal. They propose an algorithm for reconfiguration based on soft-goals contribution where they associate priorities (low, medium, high) to soft-goals, and then consider positive or negative contributions from tasks to soft-goals. For every configuration, they compute a weighted sum to privilege high-priority soft-goals.

Salehie and Tahvildari propose a mechanism for action selection in self-adaptive software based on weighted voting [ST07]. Given a goal set representing current system goals, an adaptation action set, and an attribute set, the problem they address is how to select the most appropriate action to satisfy goals in different conditions of attributes. Central element of their approach is the Goal-Action-Attribute Model (GAAM), which represents the three main abstractions and relates them in a well-defined structure. Goals are characterized by an activation level—active goals are eligible to participate in the action selection process—and a preference vector representing goal weights or priorities. System attributes are divided into controllable and non-controllable. Adaptation actions have pre- and post-conditions. They define three matrices to relate these three elements: an impact matrix shows how goals relate to actions; an activation matrix tells how goals are activated by attributes; an aspiration level matrix expresses the desired level of attributes for each goal. On the basis of this conceptual model, they apply a weighted voting mechanisms to determine the best adaptation action in the current situation.

Bryl *et al.* consider socio-technical systems able to reconfigure at runtime [BG06]. Their work is founded upon their work on the design of technical systems [BGM06, BGM09]. They observe that in modern settings the interplay between social and technical components is stronger and stronger, and alternative requirements models have to be evaluated and selected finding a right trade-off between the technical and social dimensions. They propose a planning-based approach to generate these alternative. Their conceptual model characterizes socio-technical systems as a set of actors each requesting and providing goals, and a set of possible dependencies that can be established between actors. Goals are AND/OR decomposed to sub-goals. These elements define the goals to be achieved by the system (the goals requested by individual actors) and the possible ways to achieve these goals (using capabilities for goals, decomposing them, or delegating to other actors). Formally, they translate the alternative selection problem to an AI planning problem, using the Planning Domain Description Language (PDDL), and select the configuration that costs the least. This framework has been used in [BG06] to support redesign at runtime. First, they define a set of adaptation triggers that stimulate the reconfiguration process: an agent commits to a goal, a goal is achieved, a goal is removed, an agent leaves the system, a new agent is introduced, and a new goal is introduced. Depending on the fired trigger, different preliminary operations are performed to update the problem definition, and the planner is executed to perform replanning. Costs are assigned to goal decomposition, goal delegation, and goal satisfaction via capabilities. The cost for the new strategy includes only the costs for the elements that do not belong to the current configuration. The main limitation of their approach is that they adopt a centralized planning approach, which violates agents' autonomy and heterogeneity.

Many approaches support the design of requirements-driven self-adaptive software. Lapouchnian *et al.* [LYLM06] analyse the design of autonomic application software. They demonstrate how well goal-oriented requirements engineering fits with this setting, thanks to its support to variability. Then, they show how goal models can be transformed into high-variability software designs. Finally, they describe how to use goal modelling to express self-configuration, self-optimization, and self-healing. Penserini *et al.* [PPSM07] extend the Tropos methodology to enhance its capability to support high-variability software. The proposed extensions include explicit modelling of alternatives, the adoption of an extended notion of agent capability, and a refined Tropos design process. Morandini *et al.* [MPP08] propose another extension of Tropos to support the development of self-adaptive systems. They add the notion of goal type (maintain, achieve, perform), environment modelling (including the relation between environmental and system entities), and fault modelling (representing undesirable states and describing how to prevent reaching these states). They developed a support tool, *t2x*, which generates JADEX [PBL05] agents starting from the extended Tropos models. Cheng *et al.* [CSBW09] propose goal-based modelling of adaptive systems that takes into account environmental uncertainty.

They observe that, at design-time, designers cannot know all possible environmental conditions the system will encounter at runtime. Their approach is based on the RELAX language, which integrates environmental uncertainty with goal specifications. They suggest systematic usage of tactics for adaptation to deal with uncertainty on a rising scale of costs. Examples of these tactics are adding low-level goals, relaxing requirements to accomplish partial satisfaction, identify a new (high-level) goal to mitigate uncertainty.

Baresi and Pasquale [BP10] propose an approach that exploits goal models for adaptive service compositions. They observe that very few existing approaches consider what capabilities are needed to adapt and when they should be activated. They propose an approach that extends KAOS with the notion of adaptive goal. Adaptive goals are responsible for the actual adaptation and evolution at runtime; they specify countermeasures to address violations of conventional goals. In their approach, goals are live abstractions that change dynamically. Adaptive goals describe possible adaptation strategies, and identify both the conditions that may cause a goal violation and the set of possible countermeasures. The adaptation strategy has an objective, which can be enforcing a substitute goal, the original goal, avoiding a goal violation, and enforcing a weaker version of the failing goal. Both adaptive and conventional goals are then translated into two models: conventional goals are mapped to a functional model, that provides the actual functionality of the system, whereas adaptive goals are translated to a supervision model, that is in charge of adaptation. The same authors extend their framework and propose FLAGS (Fuzzy Live Adaptive Goals for Self-adaptive systems) [BPS10]. The extension distinguishes between crisp goals with boolean satisfaction value and fuzzy goals whose satisfaction is specified via fuzzy constraints. Whereas crisp goals are defined in KAOS, fuzzy goals exploit a fuzzy temporal language inspired by the theory of fuzzy sets [Zad65].

2.2.6 Adaptation algorithms and policies

A fundamental activity in adaptive control loops is the choice of a new configuration that responds to failure or under-performance. On one side, sound and efficient algorithms are needed. On the other side, these algorithms are used in complex adaptation policies defined by system administrators. Research in this area is very heterogeneous and is often solution-specific. However, most approaches exploit the concept of resource: they suppose that software has to manage some resources, and these resources might be shared by different systems.

Wang and Li [FFZ05] acknowledge the importance of decision mechanisms in autonomic computing. They propose an autonomic computing model and a decision-making algorithm designed for that model. In their approach, a fundamental role is played by the resources managed by autonomic elements. They hypothesize that different autonomic managers need to execute operations on managed resources. Each autonomic element has a set of behavioural rules that define their behaviour. These rules describe which operations, and under what conditions, an autonomic element should execute. The proposed decision-making algorithm is executed by the overall system, and defines the best sequencing of operations to perform by the autonomic elements.

Patrascu *et al.* [PBD$^+$05] analyse resource allocation. They conceive an autonomic system as one that optimizes the way resources are allocated. They propose an efficient regret-based approach and define heuristics to further boost performance. In their approach, a central provisioner queries autonomic elements for samples of their utility functions at various resources levels. The provisioner allocates resources based on these samples. Since they consider partial utility information, they exploit the notion of minimax regret to determine a suitable allocation.

Tesauro [Tes07] propose an approach based on utility functions. His paper is a manifesto about the usage of Reinforcement Learning (RL) in autonomic computing. The key feature of RL is to require less built-in domain knowledge than traditional approaches. The basic operation of RL is to let agents learn effective decision-making policies through an iterative online trial-and-error process. In every iteration, the agent interacts with the environment: (i) it observes the state of the environment; (ii) it performs a legal action according to the observed state; (iii) it receives a reward (a numerical value it wants to maximize) and an observed transition to a new state. Tesauro shows how RL can be applied to resource allocation in a data center, where the autonomic computing resource arbiter module collects data and computes optimal allocation of data servers to applications. Unlike most decision-making algorithms, RL is not based on predefined information. As such, it is applicable to settings where design-time information is lacking or unreliable.

Many researchers define languages for adaptation policies. Policies are used by system administrators; thus, they should be expressed at a level of abstraction that they can understand without deep algorithmic knowledge. Anthony [Ant06] presents a generic policy toolkit that enables the definition of policies and their adaptation over time. In his approach, policies (i) define self-management properties for the target autonomic system; and (ii) are adaptive themselves, i.e. they can evolve over time to guarantee better adaptation of the system. The first contribution of the paper consists of an XML-based policy definition language, which allows for the specification of detailed adaptive policies. The second contribution is a policy library, which aims to support easy integration with legacy code, is thought for non-autonomics-experts, and has general applicability.

Kephart and Walsh [KW04] introduce an unified framework to define adaptation policies. Their framework supports three types of policy: (i) action policies dictate the action that should be taken whenever the system is in a certain state; (ii) goal policies declare either a desired state that should be met or criteria that maximize a set of states; and (iii) utility function policies are objective functions that express the value of each possible state. Utility functions are a generalization of goal policies, that ascribe a value to each state rather than expressing desirable and undesirable states. Figure 2.6 shows the relation between these three types of policies. In addition to showing their different level of abstraction, the figure specifies possible techniques to derive lower-level policies from higher-level ones.

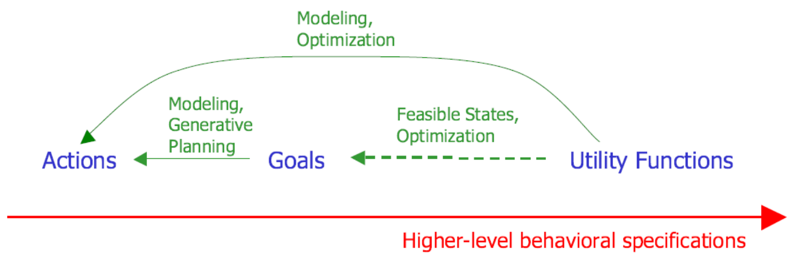

Modeling, Optimization
Modeling, Generative Planning
Feasible States, Optimization
Actions
Goals
Utility Functions
Higher-level behavioral specifications

Figure 2.6: Relation between the different types of policies defined in [KW04]

2.2.7 Agent reasoning and planning

Agent reasoning is a very relevant research for self-adaptation. Agents are computational abstractions that act in an autonomous way. Their actions can either directly affect the physical environment where they are situated (see, e.g., AI planning-based approaches), or consist of messages that are sent to other agents. The crux of agent reasoning is to determine how an agent should act to achieve its goals. Also, agent reasoning supports adaptation in response to threats to the agent's goals.

Thangarajah *et al.* [TWPF02] focus on resource conflicts in rational agents. They claim that rational agents should avoid resource conflicts while pursuing their goals. They propose a set of algorithms to derive and update resource requirements, to detect conflicts, and to resolve resource conflicts. In this approach, agents have a plan library, where each plan indicates (i) the goal for which the plan is relevant; (ii) a context precondition; and (iii) a plan body specifying what the plan does (actions and sub-goals). A particular type of action is delegation to another agent, which establishes a social relation between the delegator and the delegatee. Resource requirements are characterized as a pair ⟨resource type, amount⟩, where "amount" represents the needed quantity of that resource type. Resources are attached to plans, so that resource requirements for a certain goal can be arithmetically computed. Given a set of goals to achieve, a plan set for those goals can be safe, schedulable, schedule-dependent, conflicting, or uncertain.

Wooldridge and Parsons provide a comprehensive account on intention reconsideration in BDI agents [WP99]. They investigate how to design an agent that successfully balances the time spent in reconsidering its intentions and that spent in acting to achieve these intentions. This trade-off is a crucial issue when developing agents according to the Belief-Desire-Intention paradigm [RG92, RG95]. Intention reconsideration is very relevant to self-adaptive software, for such activity corresponds to determining whether software should adapt or persist with the current intention. Their approach is founded on the concept of meta-level control function, a policy function that determines whether the agent should reconsider its intentions and deliberate or rather go ahead with the current actions. They characterize agents in terms of four components: a next-state function, a meta-level control function, a deliberation function, and an action function. They suggest the use of utility functions to guide decision-making.

Sardina and Padgham [SP07] propose a BDI agent-oriented programming language that focuses on the semantics of goals in the presence of plan failure. Their language brings together three types of goals: classical BDI event goals, declarative goals, and planning goals. In their approach (i) agents are prevented from blindly persisting with a certain goal if a viable alternative strategy to achieve a higher-level goal exists; (ii) goals that succeed or are considered as impossible are dropped; and (iii) there is a mechanisms to proactively adopt new goals in addition to the traditional reaction goal adoption. The authors formalize mechanisms that are typical in human rational behaviour. They avoid blindly persisting with a certain goal is by integrating hierarchical planning in their agent architecture. Hierarchical Task Networks (HTNs) [Sac77] are a way to structure plans in a hierarchical manner, where high-level tasks are decomposed to lower-level ones. A very successful procedure to perform HTN task planning is UMCP [EHN94].

Myers propose the Continuous Planning and Execution Framework (CPEF) [Mye99]. CPEF starts with the generation of a rough plan. Then, such plan is executed, monitored, and repaired whenever needed. CPEF is thought for dynamic environments, where traditional planning is not sufficient to guarantee that system goals will be achieved. Therefore, plans are dynamic and open-ended artefacts that evolve over time. In CPEF, users play an important role, for they can provide feedback that influences system behaviour. CPEF is agent-based, and relies on an agent infrastructure that supports the management of complex and distributed planning tasks. It supports not only direct plan execution by software components, but also indirect execution by humans. In the latter execution mode, CPEF tracks human execution rather than carrying out the plans. Whenever the current plan has to be modified, CPEF preserves plan stability, i.e. it chooses the plan that differs the least from the current one.

Van der Krogt *et al.* [vdKdWW03] propose a resource-based framework for planning and replanning. It is founded upon the concepts of action and resource: actions produce and consume resources. They conceive both planning and replanning as transformation operations on plans. For our purposes, the most interesting part in their framework is replanning, which transforms an initial inadequate plan into an adequate plan for the fulfilment of the agent goals. They support two replanning operators: (i) plan addition glues two plans together by connecting input resources to output resources; (ii) plan deletion removes from the first plan the actions in the second plan (which has to be a sub-plan of the first one). They also propose a template algorithm for replanning, which refines a partial plan in a suitable way to achieve the current agent's goals.

A similar effort by the same authors deals with plan repair based on traditional planning [vdKdW05]. They observe that repairing an existing plan is more efficient than planning from scratch, while planning techniques are more efficient than plan repair algorithms. They manage to apply methods from traditional planning due to the operators in plan repair: (i) removal of obstructing constructs from a plan; and (ii) adding actions to achieve the goals. Whereas adding actions is easily mappable to planning, the authors propose a heuristics to deal with the removal of obstructing constructs (unrefinement). They demonstrate applicability by extending the VHPOP planner [YS03], and show experimental results that confirm the effectiveness of the approach.

Unruh *et al.* address failure handling in agents via semantic compensation [UBR04a, UBR04b]. Such technique consists of cleaning up failed or cancelled plans so that agents can behave more robustly. They describe a methodology that decouples failure handling code from normative agent logic. Semantic compensation has several benefits: (i) agents are left in a state in which future actions are more likely to succeed; (ii) agent systems are maintained in a predictable state; (iii) it is applied more generally than "patch" methods; and (iv) it enables to split a long transaction into shorter ones. Specifically, they analyse goal-based semantic compensation, i.e. compensations are defined in terms of goals. They also support compensations for tasks delegated to other agents, and they assert the need of flexible interaction protocols to achieve a useful degree of control.

2.2.8 Self-organization

Most approaches described so far conceive adaptivity as the capability of the system itself to modify its behaviour to better achieve its objectives. Self-organizing systems exploit a different approach. The behaviour of the overall system is defined by that of its components, whose objectives are not necessarily related to those of the overall system. Approaches based on self-organization are specific for application settings that resemble natural phenomena.

In his seminal work, Van Dyke Parunak [Par97] argues that agent architectures need to organize themselves and to adapt dynamically to changing circumstances without a top-down control. In his view, there is no need to develop further approaches that try to emulate human behaviour; rather, interesting theories come from natural systems, such as populations of insects. He shows several examples of multi-agent systems that exist in nature: path planning of ants, brood sorting of ants, nest building of termites, task differentiation of wasps, flocking of birds and fish, and surrounding prey of wolves. Then, he proposes a set of principles to design self-organizing multi-agent systems: (i) agents are things, not functions; (ii) agents must be kept small; (iii) system control is decentralized; (iv) agent diversity is a value; (v) an entropy leak should be provided; (vi) agents need to share information; (vii) agents plan and execute concurrently.

Mamei and Zambonelli [MZ04] propose TOTA (Tuples On The Air), a middleware to support self-organization in multi-agent systems. They observe that self-organization is founded on two building blocks: (i) adaptive and uncoupled interaction mechanisms and (ii) context awareness. The distinctive feature of TOTA is to rely on spatially distributed tuples for both building blocks. Single agents can add (inject) tuples to the network, so that other agents can exploit the published contextual information. The TOTA middleware is responsible for propagating tuples on the basis of application specific patterns, and makes sure that the intended shape is maintained. The concept of "shape" in TOTA corresponds to the values that are propagated to the various nodes in the network due to information injection. Consequently, every agent has a different perception of the environment, which depends on the nodes in the neighbourhood.

Di Nitto *et al.* [DNDM09] discuss the usage of self-organization algorithms to develop evolvable systems. They consider bio-inspired self-organizing algorithms, which are based upon a set of principles that hold in the natural world. The basic principles are (i) noise, the usage of perturbations that move the system away from its short-term goal but augment the likelihood to attain longer-term goals; (ii) emergence, the capability of a composite system to achieve a certain goal through a set of components apparently unrelated to that goal; (iii) diffusion, a method for spreading information across nodes regardless of the destination of the message; (iv) stigmergy, spreading information by storing it in the environment; and (v) evolution, the capability of the system to improve itself through natural selection of its components. They show a number of stochastic algorithms that exploit these principles. Being stochastic, there is no guarantee that they will work. However, one can reason about the probability that they will succeed.

Di Marzo Serugendo *et al.* [DMSGK06] provide a formal definition of self-organization and related concepts. They view self-organization as the mechanism or the process that enables a system to change its organization without explicit external control at execution time. They distinguish between strong and weak self-organizing system; they differ in the existence of an explicit internal central control, which exists only in weak ones. After thoroughly discussing the main properties and characteristics of self-organizing systems, they detail the property of emergence. Their analysis leads to the conclusion that emergent phenomena are characterized by (i) novelty, for what emerges is radically different from the individual components; (ii) irreducibility, i.e. the emergent phenomena exhibit properties that its components do not exhibit; (iii) interdependency between levels, for there is a relation between the evolution of lower-levels (e.g. individual agents) and the higher-levels (e.g. the multi-agent system); and (iv) an emergent phenomenon derives from non-linear activities carried out at the micro-level.

2.3 Chapter summary

In this chapter, we have provided a comprehensive survey on the state of the art in the area of self-adaptive software. In Section 2.1 we reviewed the most important models that can be exploited as a baseline for model-driven self-adaptation. These models formally represent requirements (Section 2.1.1), social interaction (Section 2.1.2), and software variability (Section 2.1.3). In Section 2.2 we detailed approaches to adaptive software.

The main contribution of this chapter is to investigate existing approaches to determine if they can be used or adapted in our approach. Section 2.1 helps us to identify the baseline for our approach, which consists of (i) Tropos goal models [BPG$^+$04]; and (ii) social commitments [Sin99]. Tropos goal models are adequate to represent STSs since they promote an agent-centric world-view. An agent adopts intentions to achieve its goals and depends on other agents for goals it cannot achieve by itself. Commitments formalize the notion of dependency in terms of a purely social abstraction that arises from interaction.

The literature review presented in Section 2.2 shows how existing approaches are inadequate to build self-adaptive software for STSs. Most approaches ignore the social component of a system, and focus on technical components only. Also, they do not consider the autonomy and heterogeneity of participants in an STS, and conceive adaptation as components re-routing, activation of different features, or orchestration of services.

Chapter 3

Variability: the key for adaptation

In this chapter, we examine the notion of variability and justify why it is the key for adaptation. First, we investigate the concept of variability in different disciplines. We start from its general meaning and conclude with software variability. Second, we relate software variability to self-adaptive software by defining the basic terms used throughout the thesis. Third, we focus on two classes of software variability: contextual and social. For each of them, we present a requirements modelling framework. The chapter provides the conceptual baseline for our approach.

Acknowledgement. Section 3.3 builds on top of on [ADG08b, ADG08a, ADG09, ADG10], while Section 3.4 extends [CDGM10b, CDGM10b].

3.1 Variability: from nature to software engineering

The notion of "variability" refers to the state or characteristic of being variable. According to the Merriam-Webster online dictionary[1], variable means "able or apt to vary; subject to variation or changes". Consequently, the term "variability" points to a non-static nature and to the ability to change. The same word is used to denote specific phenomena in various fields: climate, genetic, heart rate, space, statistics, product engineering, etc.

The best-known acceptation of "variability" is perhaps that in genetics. Genetic variability is a natural phenomena that describes the tendency of the genotypes in a population to vary from one another. Genetic variability is fundamental for biodiversity: without variability, a population can hardly adapt to environmental changes and becomes more exposed to the risk of extinction. This specific meaning influences the understanding of variability in non-natural domains, such as product engineering and software engineering. In these fields, variability enables the creation of customized variants of a same prod-

[1]http://www.merriam-webster.com/

uct/software that better adapts to user needs and preferences. Unlike genetic variability, product/software variability is an intended effect, and not a natural phenomenon.

Product lines have a long-standing tradition in industrial manufacturing. A product line is "a group of products that are closely related because they function in a similar manner, are sold to the same customer groups, are marketed through the same types of outlets, or fall within a given price range" [KA09]. Product lines exploit product variability: a generic product is designed so that different product variants—each differing for some feature—can be produced. An example of product variants are the different sizes and colours T-Shirts can come with: each combination of size (small/medium/large/x-large) and colour (red/blue/yellow) is a different variant. An interesting case study about product variability concerns car bodyworks[2]. The same bodywork is exploited to produce very diverse car variants: sedan, station wagon, convertible, and pickup. The main advantage of product variability is to reuse a single design for different products to accommodate more customers than a single unchangeable product would do.

Product lines have been applied to software production, and are known as software product lines or software product families. Software product lines are based on the same rationale as that of traditional product lines: a single design can be exploited to deploy multiple variants of the same software. Variants differ in terms of the features they provide and, consequently, of price. Software variability is often expressed using feature models [KCH$^+$90, KKL$^+$98]. Feature models are trees whose nodes are features and whose links are relations between features. A feature can be decomposed to an arbitrary number of sub-features, some features can be optional, a feature can be decomposed to alternative features. Software variability is used not only in solution-space artefacts such as feature models, but also in problem-space artefacts such as goal models [LLY$^+$06, LJL$^+$07].

3.2 From variability to self-adaptive software

Self-adaptive software is a class of software systems able to autonomously vary its behaviour in response to changes in the environment. Variability is essential for self-adaptive software: without variability there would be only one variant, thus software could not adapt. Software adaptation is inevitably linked to requirements: software adapts to better achieve its requirements, when they are threatened. Let's provide a more formal account for these intuitions.

Definition 3.1 (Software Variability) *The property of a software system to have alternative variants among which choosing.*

[2]http://mercedes-benz-blog-trivia.blogspot.com/2010/02/mercedes-benz-blog-trivia-four-cars.html

The provided definition of software variability is intentionally general. First, we do not prescribe *who* actuates the change. Thus, the definition allows both for human intervention or autonomous change. Second, variability is a structural characteristic: software should be designed with the characteristic of being variable. Third, the definition of variability does not specify exactly what is the object of change; thus, the definition applies to component-based systems, service-oriented applications, agent-based software, and so on. From now on, we will use the term "variability" to refer to "software variability". The definition of variability references the concept of variant, which is defined below.

Definition 3.2 (Variant) *An operational configuration of software.*

The definition of variant characterizes software operation in general terms, without binding it to any technology. In our approach—and in literature—variants are the concrete manifestations of variability. Variability exists if and only if there is more than one variant. If there is only one variant, then software will have only one operational configuration, and it will not be changeable. If more variants exist, software is ready to be changed.

Definition 3.3 (Adaptation) *The process through which software switches from the current variant to a different one to better fulfil its current requirements.*

The concept of adaptation we introduce here is more specific than those of variability and variant. Let's emphasize the salient non-obvious points in this definition. First, adaptation involves variants, and in particular it is the transition from the current variant to a different one. Thus, staying with the same variant is not an adaptation. Second, adaptation is related to requirements. Adaptation is performed to better fulfil the current software requirements. This means that the adoption of a variant that does not *better* fulfil the requirements—or does not fulfil them at all—is not an adaptation.

How does our characterization relate to natural adaptation and evolution? According to Eldredge and Hurst [Eld95], "adaptation is the heart and soul of evolution". Modern evolutionary theories seem to argue that adaptation via genetic change leads to exemplars that are not necessarily better adapted to the environment than previous ones. In our case, term "adaptation" refers to ameliorating transitions. Software can clearly switch to a worse variant, but that is not an adaptation according to Definition 3.3. We made this choice since software, being a human artefact, can be designed to perform as good as possible (designing for under-performance makes little sense).

Definition 3.4 (Self-Adaptation) *An adaptation process autonomously carried out by software.*

Definition 3.5 (Self-Adaptive Software) *Software capable of self-adaptation.*

The definitions of self-adaptation and self-adaptive software follow from the previous ones. Self-adaptation refers to a particular kind of adaptation that is autonomously carried out by software, instead of being performed by some external actor (typically, humans). Self-adaptive software represents the family of software systems capable of self-adaptation.

Thanks to this set of definitions, the reader should have a clear understanding of the core concepts concerning software adaptation. We clarified the relation between variability, variants, adaptation, and self-adaptive systems. To demonstrate the flexibility of these definition, we present a few examples of diverse self-adaptive applications that can be characterized using Definitions 3.1-3.5:

– suppose a *distributed DBMS* is deployed on a network of servers, and it allocates queries and maintenance jobs to different servers depending on their current workload. So, given the set of current jobs, each possible allocation of jobs is a different variant. The main requirement of the DBMS is to guarantee that all jobs are carried out as efficiently as possible. Adaptation consists of (i) migration of jobs from one server to another; (ii) addition of new maintenance jobs to optimize low-performing machines; and (iii) efficient allocation of new queries. The DBMS performs self-adaptation when these operations are conducted by the DBMS itself;

– suppose one of the requirements of a *smart-home* is to assist the patient in preparing breakfast. Different variants are possible: using cameras to monitor the patient's activity, calling a catering service and facilitate delivery by opening the door, showing visual aids to guide the patient during breakfast preparation. Adaptation means switching from one variant to another if breakfast preparation is in danger. For example, if the cameras reveal the patient is idle, then the smart-home might intervene by guiding the patient via visual aids;

– consider a *service application* that provides real-time meteorological information to emergency response units in the county. Its main requirement is to provide current weather and accurate 7-days forecasts. Also, data should be provided in a redundant way, i.e. relying on different information sources. The service application gathers information from different web services. Given the set of available web services providing the needed data, each combination that includes redundant information provision is a variant. Adaptation means dropping some service and replacing it with another one, if the current service is unreliable.

These examples show that the definitions are able to deal with a variety of threats (low-performance, plan failure, unreliability) through diverse reconfiguration actions (job reallocation, usage of effectors, service replacement). Also, they show how adaptation is performed to better meet requirements: to carry out jobs efficiently, to ensure the patient has breakfast, to provide redundant weather information. In the next sections we will construct our framework for software self-adaptation on top of Definitions 3.1-3.5.

3.3 Contextual variability

The context where software is deployed impacts on software operation. Changes in the context may threaten the capability of software to meet its requirements (i.e. the current software variant might become ineffective). Software operation has an impact on context as well. While achieving its requirements, software changes the context. Thus, the influence between requirements and context is mutual. We propose a requirements modelling framework, the contextual goal model, that allows for representing and reasoning about such influence.

3.3.1 On the notion of context

According to the Merriam Webster online dictionary, context is “the parts of a discourse that surround a word or passage and can throw light on its meaning” or “the interrelated conditions in which something exists or occurs”. The former definition is related to linguistics, whereas the latter describes the ontological meaning of context. The same dictionary suggests two synonyms for such word: “environment” and “setting”. This generic definition emphasizes how anything existing or occurring should be understood as part of a wider entity, its context.

Literature in artificial intelligence provides many definitions of context. As Brezillon observed [Bre99a, Bre99b], there is no agreement on the meaning of the term. In McCarthy’s notes on formalizing context in AI [McC93], context is “an infinite and only partially known collection of assumptions”. The Telos knowledge representation language [MBJK90] define it simply as “possible worlds”. Giunchiglia gives a definition suited for contextual reasoning [Giu93]: “a context is a theory of the world which encodes an individual’s subjective perspective about it”. These definitions are flexible and well-thought; however, they do not tell which are the actual elements that characterize context. Such missing information is valuable for system engineers.

More recently, researchers in ubiquitous, mobile and pervasive computing have proposed their own understanding of context. The best-known definition is probably that by Dey [Dey01]: “Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.”. Such definition is recursive, for it refers to a so-called “situation” of an entity.

As such, it does not help much, since defining a “situation” is a hard task too.

Context plays an important role in requirements engineering. Researchers in this area often use different terms to refer to context, such as environment. In their seminal paper [ZJ97], Zave and Jackson argue that requirements exist only in the *environment*, a portion of the real world whose current behaviour is unsatisfactory in some way. They claim all statements made in requirements engineering are about the environment, none is about the proposed machine. This notion of environment is very similar to the definitions of context we have listed above. Such tight relation is confirmed by Finkelstein and Savigni [FS01]: they assert that context is “the reification of the environment”.

The problem frames approach [Jac00] is based on the centrality of the environment. It characterizes software problems along three dimensions: (i) the world description W expresses unconditional behaviour and properties of the world that do not depend on the designed machine, (ii) the requirements R express the desired behaviour and properties the world should exhibit as a result of the interaction with the machine, and (iii) the specification S describes the behaviour and properties the machine should exhibit in its interface with the world.

3.3.2 Context and goals

As shown in Section 3.3.1, there is no agreement on the definition of context. Our intent is not to provide an universal definition of context. Rather, we will provide a pragmatic definition that captures its tight relation with requirements. Consequently, our definition should be applicable to the requirements modelling language we choose.

Since goal models represent variability at the level of requirements [LLY$^+$06, LJL$^+$07], they are a suitable option for our purposes. Though traditional goal modelling approaches do not explicitly represent context, designers implicitly consider such factor when they introduce variation points (e.g. OR-decompositions) in goal models. Let’s show this intuition on a smart-home scenario. Suppose the patient’s goal is to “Have breakfast”. Possible alternatives to achieve such goal are “Prepare breakfast autonomously” and “Order catering food”. Using a goal model, a designer would represent these two alternatives as sub-goals of the OR-decomposed goal “Have breakfast”. The designer might have made implicit contextual assumptions: (i) the first option is applicable only if the patient is autonomous (i.e. he’s not constricted in bed); and (ii) the second option is applicable only if the patient is able to contact a catering service.

As said in the conclusions of Chapter 2, our baseline consists of Tropos [BPG$^+$04] goal models. Tropos describes a system as a set of interdependent actors, each having its own goals to reach. Tropos goal analysis decomposes high-level objectives into lower-level ones. Goal analysis ends with the introduction of executable tasks. A task is linked to a goal by a binary "means-end" relation: the execution of a task is a means to achieve a goal. We extend Tropos to represent the mutual influence between requirements—understood as goals—and context. We introduce now the basic concepts of our framework (from [ADG10]).

Definition 3.6 (Actor) *An actor is an entity that has goals and decides autonomously how to achieve them.*

The notion of actor accommodates different types of *active* entities, such as humans, software, or organizations. An actor is characterized by two fundamental properties. First, actors are goal-oriented: they have certain goals that they want to achieve. In general, actors do not disclose their goals to other actors (goals are kept private). Second, actors are autonomous: they make decisions on how to achieve their goals, and others cannot force them to act in a specific way. A corollary of these two properties is actor heterogeneity: each actor has its own goals and devises its strategies to achieve them.

Let's illustrate these concepts on the mall scenario. Suppose a sales staff—a human actor—has the goal of conveying information about products to customers. The sales staff is free to decide when to pursue (activate) such goal and which are possible strategies to reach it. Due to actors heterogeneity, such strategy might be invalid from the perspective of a different actor. The staff may reach such goal in different ways: he can (i) call the customer by phone, or (ii) deliver information to him in person. The sales staff himself is the only actor that can select which strategy to adopt. Such decision is influenced by the state of that portion of the world where the actor is. We call such state "context".

Definition 3.7 (Context) *A context is a partial state of the world that is relevant to an actor's goals.*

We devise a definition of context is expressly suited for actor-based conceptual frameworks in which actors are goal-oriented entities (as in Definition 3.6). Hence, it should not be taken as a general definition of context. Its specific features are the following:

– *Actors observe the world purposefully*: there is no value for an actor to observe the world for the purpose of observation per se. Actors do that to guide decision making about which goals to reach and which actions to perform for reaching those goals. Such decisions are influenced by properties of the world in which the actor lives. For example, a partial world state like "customer is interested in a product" is relevant for a sales staff whose goal is to promote a product to customers. The same context is irrelevant if her goal is to process customer complaints.

– *State of the world relevancy is subjective*: the decision about which are the parts of the world relevant to decision making is of subjective nature. Each actor has a viewpoint on the world and is free to choose which are the relevant states of the world. For example, to decide whether "conveying information to a customer via an information terminal" is adoptable, one sales staff S_1 might observe the context "customer is very close to a free terminal", whereas staff S_2 might verify the context "customer visitor has a map showing where terminals are located".

– *Context is inherently partial*: actors have only a partial view on the state of the world. They are neither interested nor able to capture all the information that fully describes such a state. For instance, a sales staff that wants to promote soccer jerseys is probably not interested in recently released philosophy books. The same sales staff, while promoting a jersey, is unable to check if there is a traffic jam nearby his house.

– *Context is volatile*: the world is volatile and could be in different states. An uniform partial state of the world is irrelevant for the decision making of an actor. For example, if a promotion information system operates in a geographic area where shopping malls do not provide information offices, then the system need not consider their existence when deciding about how to convey information to a customer. However, many states of the world are volatile. For example, the interest of a customer in a certain product changes—and sales staff should promptly exploit the interest before it ceases; also, the number and identity of customers in the mall varies.

3.3.3 Contextual goal model

The definitions given in Section 3.3.2 provide the conceptual background for our modelling language to represent and deal with contextual variability. We extend Tropos proposing the contextual goal model [ADG10]. The key novelty of this modelling language is to explicitly represent and analyse the relation between context and goals.

The contextual goal model enriches variation points with information about the impact of context. Such impact is expressed via contextual annotations (*contexts*). Contexts are represented as labels (c_1, c_2, ..., c_n). Every label is associated to a natural-language description of the context. In Section 3.3.4 we will show how such description is formally specified via *context analysis*. Moreover, goal labels contain parameters between square brackets (e.g. [c] for customer, [p] for product). Parameters are needed during context analysis. Figure 3.1 shows a partial contextual goal model for the promotion information system. Contextual goal models support the following contextual variation points:

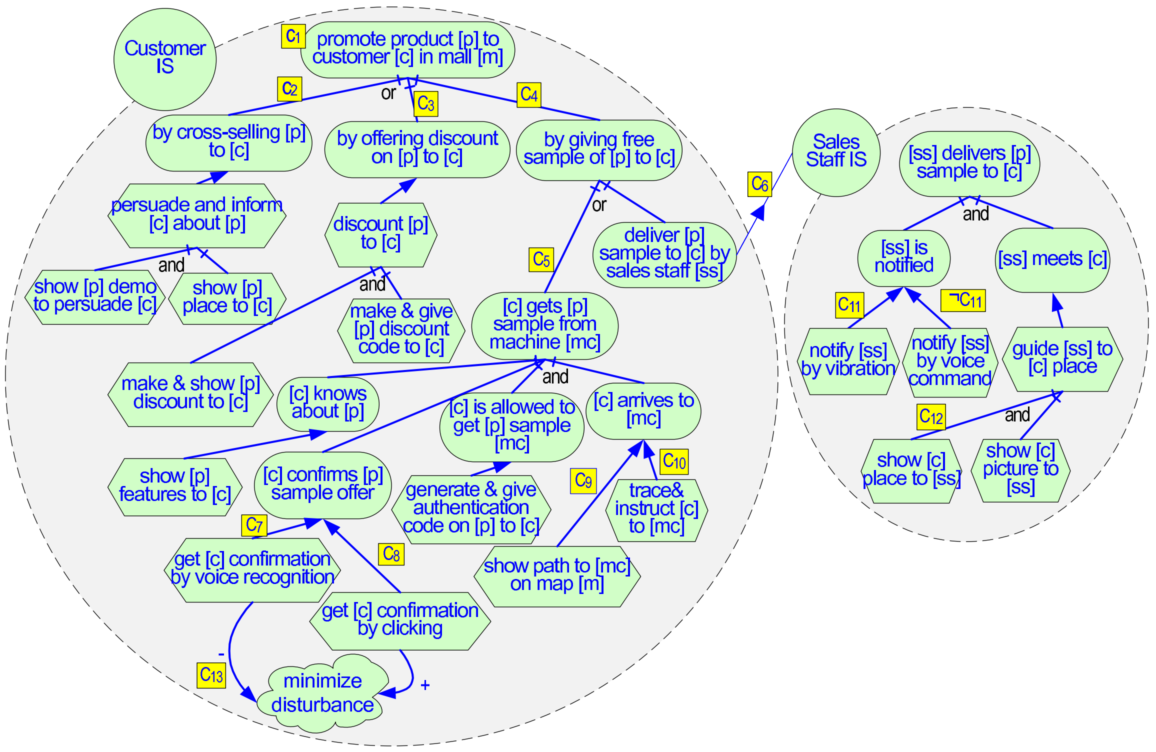

Customer IS
C1
promote product [p] to customer [c] in mall [m]
C2
or
C3
C4
by cross-selling [p] to [c]
by offering discount on [p] to [c]
by giving free sample of [p] to [c]
persuade and inform [c] about [p]
discount [p] to [c]
or
deliver [p] sample to [c] by sales staff [ss]
C5
C6
and
show [p] demo to persuade [c]
show [p] place to [c]
and
make & give [p] discount code to [c]
[c] gets [p] sample from machine [mc]
make & show [p] discount to [c]
[c] knows about [p]
and
[c] is allowed to get [p] sample [mc]
[c] arrives to [mc]
show [p] features to [c]
[c] confirms [p] sample offer
generate & give authentication code on [p] to [c]
C9
C10
trace& instruct [c] to [mc]
C7
get [c] confirmation by voice recognition
C8
show path to [mc] on map [m]
get [c] confirmation by clicking
-
C13
minimize disturbance
+
Sales Staff IS
[ss] delivers [p] sample to [c]
and
[ss] is notified
[ss] meets [c]
C11
¬C11
notify [ss] by vibration
notify [ss] by voice command
guide [ss] to [c] place
C12
and
show [c] place to [ss]
show [c] picture to [ss]

Figure 3.1: Contextual goal model for a mall promotion information system. Contexts are associated to variation points as labels

– *OR-decomposition*: the adoptability of each sub-goal (sub-task) of an OR-decomposed goal (task) may require a specific context to hold. Consider the OR-decomposition of goal promote product [p] to customer [c] in mall [m] in Figure 3.1. Subgoal by cross-selling [p] to [c] can be adopted when the product complements another product the customer already has (c_2); by offering discount on [p] to [c] is adoptable when the product is discountable and interesting to the customer (c_3); and by giving free sample of [p] to [c] is adoptable when product is free sampled and new to the customer (c_4). Goal [c] gets [p] sample from machine [mc] can be adopted when customer has experience with such machines and can reach it quickly (c_5).

– *means-end*: goals are satisfied by means of executable processes (*tasks*). The adoptability of each task may require a specific context. Consider the means-end decomposition of goal [c] confirms [p] sample offer. Task get customer [c] confirmation by voice recognition is applicable only if the place where the customer is currently not noisy, and the system is trained enough on the customer voice (c_7). The alternative task get [c] confirmation by clicking can be adopted when the customer has a good level of expertise in technology and the device has a touch screen (c_8). The task show path to sample machine [mc] on the mall e-map [m] is adoptable if the customer can arrive easily to that machine (c_9), while task trace and instruct customer [c] to sample machine [mc] is adoptable if the path is complex (c_{10}).

– *actor dependency*: the applicability of a dependency might depend on a specific context. For example, the promotion information system can satisfy goal deliver a sample of the product to customer by sales staff by delegating it to the sales staff information system, if the corresponding sales staff knows that product sufficiently well and is currently available (c_6).

– *root goals*: root goals are activated only in certain contexts. For example, to activate goal promote product to customer in mall, the customer has to be inside the mall building and be eager to accept getting promotion of the product (c_1).

– *AND-decomposition*: the satisfaction (execution) of a sub-goal (sub-task) in an AND-decomposition might be needed only in certain contexts. In other words, some sub-goals (sub-tasks) are optional to fulfil the decomposed goal (task). For example, sub-task show customer current place to sales staff is not needed if the customer can visually see the sales staff (c_{12}).

– *contribution to soft-goals*: soft-goals are qualitative objectives for whose satisfaction there is no clear-cut criteria. Mylopoulos *et al.* used the term "satisficed" [MCN92] to emphasize that, unlike hard goals, they cannot be satisfied. Tasks and goals can contribute either positively or negatively to soft-goals; contribution can be qualitative ($-$, $--$, $+$, $++$) or quantitative (a real number in the range $[-1, +1]$). The contributions to soft-goals can also vary from one context to another. In our example, task get customer confirmation by voice recognition contributes negatively to soft-goal less disturbance when the mall is crowded (c_{13}).

3.3.4 Context analysis

In Section 3.3.3 we have introduced the contextual goal model and have shown how contextual labels are associated to variation points. These labels are place-holders for descriptions of the corresponding context. However, such description is not formal. Like goals, context can be analysed too. Whereas goal analysis allows for discovering alternative sets of tasks an actor may execute to reach a goal, context analysis should allow for discovering alternative sets of facts an actor may observe to judge if a context holds.

We introduce now the basic concepts of context analysis. The first concept is that of formula of world predicates (*Form*), which is defined by the following EBNF grammar as an AND/OR composition of world predicates:

$$Form := World\_Predicate \mid (Form) \mid Form \wedge Form \mid Form \vee Form$$

There are two types of world predicates, that differ for their observability by an actor: *facts* and *statements*.

Definition 3.8 (Fact) *A world predicate W is a fact for an actor A iff A can observe W.*

Definition 3.9 (Statement) *A world predicate W is a statement for an actor A iff A cannot observe W.*

Actors observe facts via a well specified procedure. They need the ability to capture the necessary data and to compute the truth value of a fact. Facts are not viewpoints. When a fact is true for an actor it will be also true for others. The notion of fact is unrelated to that of belief, that we do not consider in our framework. Since facts are observable world predicates, they hold irrespective of the beliefs of the different actors.

A world predicate such as "customer bought products from the mall in the last week" is a fact. To verify this fact, the Customer IS actor can check the purchase history of the customer in the mall database. The world predicate "two products, p_1 and p_2, are usually sold together" is also a fact. The system can check the sales record of all customers and check if the two products p_1 and p_2 are often sold together. The world predicate "product is not in the shopping cart of the customer" is a fact observable by positioning RFID readers in shopping carts.

Not all world predicates are observable by an actor. We call these non-observable predicates *statements*. There are several reasons why a world predicate cannot be observed, among which the following:

– *lack of information*: an actor may be unable to observe a world predicate because of its inability to capture the necessary information to observe it. For example, "customer does not know about a new product" is a statement from the perspective of an actor such as the sales staff in a shopping mall. The staff cannot obtain all the information needed to observe this statement, since the staff cannot monitor if a customer has read about the product on the web or has been told about it by a friend.

– *abstract nature*: some world predicates are abstract by nature and cannot be evaluated against clear criteria. For example "customer is interested in a product" is a world predicate that cannot be observed in a well specified way. It is a concept that refers to the mood of a customer; therefore, only the customer itself can tell whether such predicate holds or not.

– *subjectivity*: the validity of a world predicate depends on the actor that evaluates it. Notice how the notion of statement is different from that of fact: whereas facts are not viewpoints, statements can be (and often are) viewpoints. For example, a world predicate like "customer is rich" can be evaluated differently depending on the criteria an actor uses to determine wealth.

Some contexts are specifiable by means of only facts, while others need the use of statements. For example, to decide whether promoting a product by offering a discount, the system (Customer IS system) has to judge if context c_3 applies. This includes deciding the truth of the world predicate *wp*="customer is interested in the product". Such world predicate is a statement that the system cannot observe. However, this statement can be *refined* into a formula of facts and other statements. For example, a possible refinement might consider the behaviour of the customer in the mall and his purchase history. If the customer has been examining the product for more than ten minutes or he often goes to the product area, then the system may judge that *wp* holds. We introduce a new relation that links a formula of word predicates to a statement. Such relation is called *Support* and enables to refine a statement.

Definition 3.10 (Support) *A statement S is supported by a formula of world predicates φ iff φ provides evidence in support of S.*

Iteratively, a statement could be ultimately refined to a formula of facts that supports it. In other words, the relation *support* is transitive. If a formula φ_1 supports a statement S_1 and $S_1 \wedge \varphi_2$ supports S_2, then $\varphi_1 \wedge \varphi_2$ supports S_2. However, refining a statement to a formula of facts is not always feasible. In contextual goal models, we allow only for contexts that are specified in terms of facts and/or statements that are supported by facts. We call these kinds of statements and contexts, *observable statements* and *observable contexts*, respectively. They are defined as follows:

Definition 3.11 (Observable Statement) *A statement S is observable iff there exists a formula of facts φ that supports S.*

Definition 3.12 (Observable Context) *A context C is observable iff C can be specified by a formula of facts and monitorable statements.*

A observable context, specified by φ, holds if all the facts in φ and all the formulae of facts that support the statements in φ are true. Notice that facts are observable by definition, since there exists a procedure for an actor to observe them. *Context analysis* allows to discover if a context is observable and to find the formula of facts that specifies it. Context analysis starts with the specification of a world predicate formula that represents a context. This formula may contain both facts and statements. Take context c_1 in Figure 3.1, and let its specification be $c_1 = wp_1 \wedge wp_2$ where wp_1="customer is inside the mall building" and wp_2= "customer may accept getting promotion of the product". wp_1 is a fact the system can observe using a positioning system, while wp_2 is a statement and needs further refinement.

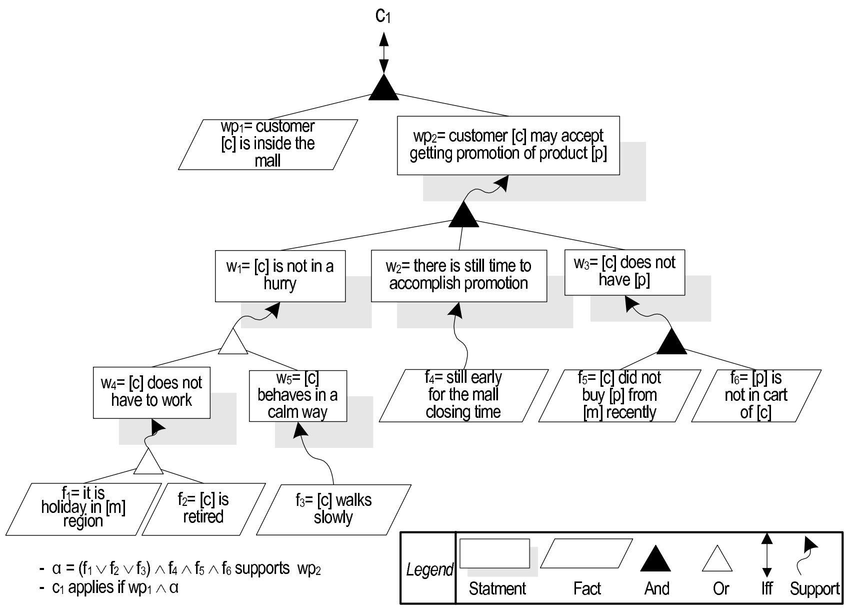

c1
wp1= customer [c] is inside the mall
wp2= customer [c] may accept getting promotion of product [p]
w1= [c] is not in a hurry
w2= there is still time to accomplish promotion
w3= [c] does not have [p]
w4= [c] does not have to work
w5= [c] behaves in a calm way
f4= still early for the mall closing time
f5= [c] did not buy [p] from [m] recently
f6= [p] is not in cart of [c]
f1= it is holiday in [m] region
f2= [c] is retired
f3= [c] walks slowly
- α = (f1 ∨ f2 ∨ f3) ∧ f4 ∧ f5 ∧ f6 supports wp2
- c1 applies if wp1 ∧ α
Legend
Statment
Fact
And
Or
Iff
Support

Figure 3.2: Context analysis for context c_1 in Figure 3.1

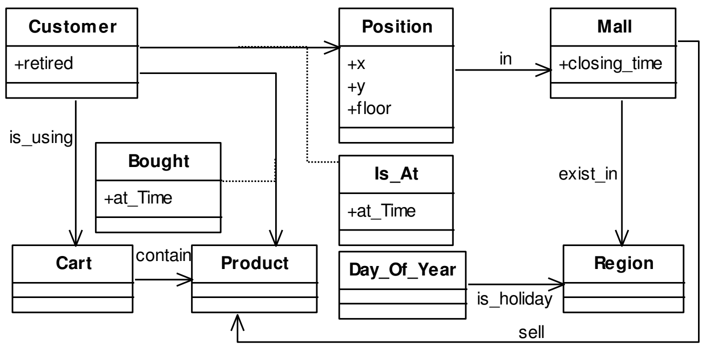


Figure 3.3: Data conceptual model specifying required data to observe c_1 (Figure 3.1)

Figure 3.2 shows our visual syntax for the context analysis of c_1. Statements are represented as shadowed rectangles, facts as parallelograms. The support relation is represented as a curved filled arrow, the and, or logical operators are represented as black triangles and white triangles, respectively. Equivalence (filled double-arrow) relates a context label to its specification in terms of a formula of facts and statements.

Context analysis enables the discovery of which data about the world an actor has to collect. Indeed, an actor can observe facts only if it can gather required data. For example, the facts in the context analysis of Figure 3.2 lead to the data conceptual model in Figure 3.3. Such model expresses the monitoring requirements for the promotion system.

3.3.5 Methodological guidance

Requirements analysts need guidance to properly conduct contextual goal modelling. Indeed, such activity consists of heavily intertwined sub-activities wherein stakeholders and analysts tightly cooperate. Devising a comprehensive methodology for contextual goal modelling is outside the scope of this thesis. A comprehensive methodological account on contextual goal modelling can be found in [Ali10]. Here, we outline the main activities that are necessary to develop self-adaptive applications based on contextual goal models. The way the core activities are intertwined is shown in the activity diagram in Figure 3.4.

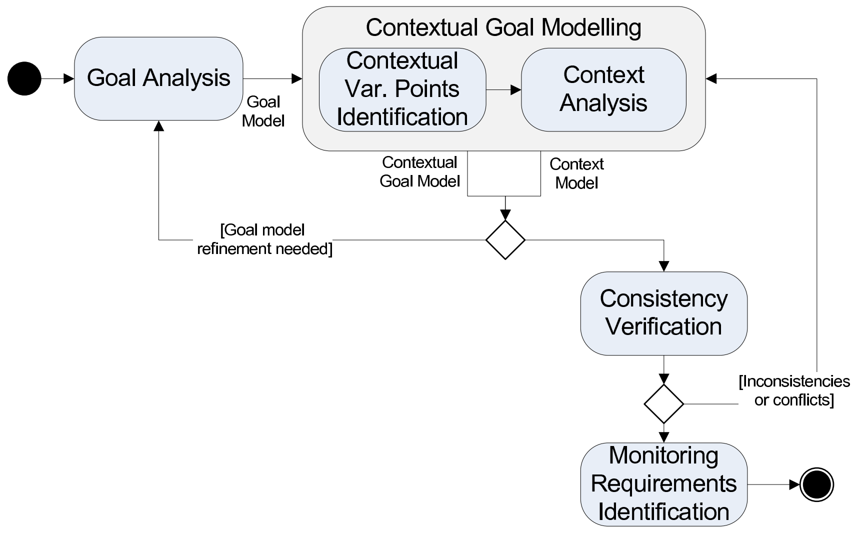

Goal Analysis
Goal Model
Contextual Goal Modelling
Contextual Var. Points Identification
Context Analysis
Contextual Goal Model
Context Model
[Goal model refinement needed]
Consistency Verification
[Inconsistencies or conflicts]
Monitoring Requirements Identification

Figure 3.4: Methodological guidance for contextual goal modelling

1. *Goal Analysis* consists of the identification of high-level goals and their refinement (goal analysis). Traditional goal modelling methodologies (e.g. the early requirements phase of Tropos) can be applied.

2. *Contextual Goal Modelling* activities are conducted after goal analysis. These activities add contextual information to goal models. Goal analysis and contextual goal modelling can be conducted iteratively. After each iteration, further goal analysis is performed if needed. We identify two sub-activities:
   (a) *Contextual Variation Points Identification*: the variation points that are affected by contextual factors are identified.
   (b) *Context Analysis*: the contexts affecting contextual variation points are analysed by applying the context analysis described in this section.
3. *Consistency Verification*: automated reasoning techniques are performed to verify the consistency of the contextual goal model (e.g. applicability of variants, identification of redundant variants, detection of inconsistent contexts). In case inconsistencies are detected, further contextual goal modelling is required. Details about consistency verification techniques are provided in [ADG10].
4. *Monitoring Requirements Identification*: once contextual goal models are consistent, the analyst has to identify the monitoring requirements for the system. These requirements consist of the data the system should collect from the system environment and the equipments needed to collect them.

3.3.6 Variation points and self-adaptive software

As we have observed earlier, goal models enable to represent variability at the level of goals. In Section 3.3.3 we have proposed the contextual goal model as a framework to represent the interweaving between context and goals. Contexts are first associated to variation points via labels, then each context is refined to a set of observable facts through context analysis (Section 3.3.4).

Context monitoring is therefore an essential feature for self-adaptive software dealing with contextual variability. Our conceptual framework can be exploited to represent context changes that inhibit the current variant and to identify new opportunities that were not previously available. The data conceptual model that is derived after context analysis represents monitoring requirements (what the system shall monitor).

Here, we explain and illustrate how the different contextual variation points affect the variability space.

– OR-decomposition: context helps to reduce the variability space, for it specifies required preconditions to adopt sub-goals. Consider the OR-decomposition in Figure 3.5 extracted from the contextual goal model in Figure 3.1. Context c_2 is required for cross selling, c_3 for offering a discount, c_4 for giving a free sample. The contexts associated to the OR-decomposition branches need not be mutually exclusive. Consequently, more than one alternative might be available at the same time. For instance, a product might be both associated to a cross-selling offer (c_2) and be discounted (c_3). It might also be the case that no sub-goal is adoptable. Suppose

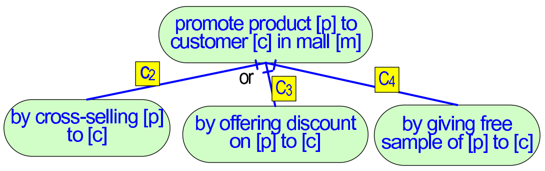

promote product [p] to customer [c] in mall [m]
or
C2
C3
C4
by cross-selling [p] to [c]
by offering discount on [p] to [c]
by giving free sample of [p] to [c]

Figure 3.5: Contextual OR-decomposition taken from Figure 3.1

now a self-adaptive promotion system decides to promote the product by giving a free sample. Also suppose that, while carrying out the plan to give a free sample, the last free sample is taken by another customer. Context c_4 becomes false—no free sample is available—and the information system will have to switch to a different variant. The system will choose on the basis of the validity of c_2 and c_3. If neither context is true, then no alternative variant exists, and the system cannot adapt.

– means-end: the role of context in means-end decompositions is analogous to that in OR-decomposition. The difference is conceptual: in a means-end decomposition context affects task execution, rather than goal adoption. In Figure 3.6 context limits the applicability of different tasks for the same goal. Showing a map is possible only if c_9 holds, while tracing and instructing the customer only if c_{10} holds. Like in OR-decompositions, it is possible that both contexts hold (all variants are possible) or that none holds (no variant is applicable).

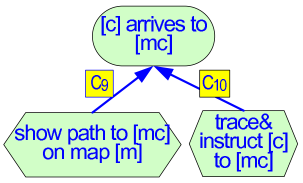


Figure 3.6: Contextual means-end decomposition taken from Figure 3.1

– AND-decomposition: the effect of context on AND-decompositions is opposite to that of OR-decompositions and means-ends. In AND-decompositions, context makes certain sub-goals/sub-tasks needed if the context associated to their decomposition branch is true. Consider the contextual AND-decomposition in Figure 3.7. If the context c_{12} holds, then both sub-tasks should be performed to execute the decomposed task. Otherwise, only the second task has to be executed. Suppose our self-

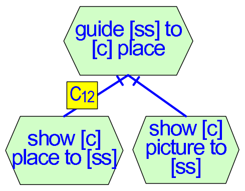


Figure 3.7: Contextual AND-decomposition taken from Figure 3.1

adaptive promotion system is guiding the sales staff to the customer place and c_{12} does not hold (the sales staff can easily locate the customer). Suppose c_{12} becomes true due to the entrance of several customers in the mall. Now, the self-adaptive system will switch to a variant including task show customer place to the sales staff.

– root goal activation: the effect of context is to specify in which context a root goal is activated. The impact on adaptation is different from that of the previous variation points. Indeed, such context does not invalidate the current variant nor makes new variants viable. The effect is that software should satisfy a new goal, thus it should revise its strategy and choose a variant supporting such goal too. Suppose the promotion system has another activated root goal issuing loyalty cards to new customers. Certain variants to achieve this goal might inhibit product promotion. For instance, assigning a sales staff to the issuing of loyalty cards could inhibit the promotion of a product to another customer.

– dependency: contextual dependencies are applicable only if the corresponding context holds. For instance, the dependency on the sales staff for delivering a free sample (Figure 3.1) is possible only if the sales staff is able to promote the product and has enough time. If our self-adaptive information system is relying on such dependency, but during its execution all sales staffs are busy with higher-priority duties (c_6 becomes false), the system should choose a different variant not including that dependency.

– contribution: contextual contribution does not affect variants applicability. Unlike other contextual variation points, it has a qualitative effect. In a certain context a variant V_1 might be preferable to V_2 because of its better contribution to soft-goals. In another context V_2 might provide better contribution. Self-adaptive software should continuously monitor contexts related to contributions to identify threats to the validity of the current variant and opportunities to better perform.

3.4 Social variability

We analyse here social variability, which is caused by the varying social structure of STSs. Specifically, it exists due to the social relationships that arise, hold, evolve, and cease to exist. This kind of variability is becoming very relevant in software engineering, due to the diffusion of computational paradigms characterized by heavy interaction among technical systems and social actors. The increasing number of Web 2.0 social networks and ubiquitous computing systems seems to support our claim.

Consequently, the design of self-adaptive software cannot ignore social variability. Software systems do not operate in isolated environments, but they execute in *open systems* together with other systems. Moreover, participating subsystems are not only software systems, but also humans and organizations. These social relations define business transactions between legal entities, for software systems act on behalf of legal entities. Let's consider eBay. When we use eBay and create an auction for an old book, we establish business relations with the eBay corporation—we commit to sell a real item and we accept the eBay policies—and with the bidders—we commit to ship the item to the winner upon payment.

We illustrate the notion of social variability with the aid of a small example about emergency response coordination. Suppose the current goal of a fire chief is to put out a fire. Figure 3.8 shows a scenario where the fire chief uses his PDA to set up a fire squad to put out that fire. The requirements of the PDA software are (i) to ensure that two fire units will reach the fire, and (ii) to make sure that the closest firemen are selected. The dashed rectangles represent an abstract view on variants; the arrows indicate the events that trigger adaptations in the PDA software. The initial variant is shown in the top-left of the figure, where the fire chief is connected to fire unit 1 and fire unit 2 via social relations; each fire unit is committed to reach the fire. The fire chief's PDA established these relations by sending an SMS to the fire units, which have responded affirmatively. After some time, fire unit 3 joins the STS. Such fire unit is the closest to the fire, therefore the software identifies an opportunity and decides to assign fire unit 3 to the fire and to release fire unit 1 from its commitment. Later, fire unit 2 violates its commitment: it communicates that it will not be able to reach the fire due to a traffic jam. In response to this event, the fire chief's PDA reassigns the task to fire unit 1, which accepts such task and establishes a new social relation. Finally, both fire unit 1 and fire unit 2 leave the system (e.g. they go out of service). The fire chief should now rely only on fire unit 3.

Where is social variability in this setting? Social variability consists of the ever-changing social structure that exists in open systems. Actors are free to join and leave as they please, they offer different services at different times, they establish, maintain, drop,

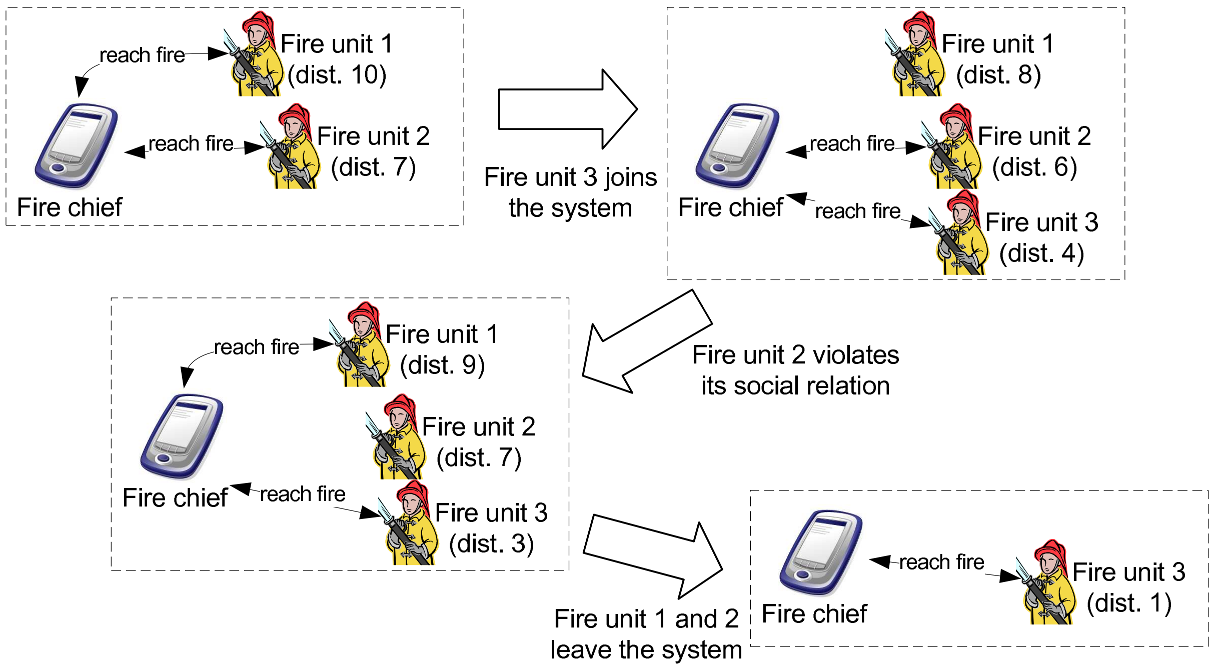


Figure 3.8: Social variability and adaptation in an STS concerning fire-fighting

and release social relationships. Therefore, the social context where a system operates is in constant evolution, and the system should continuously monitor such context to detect threats and to identify new opportunities.

Let's contrast social variability to contextual variability. Contextual variability is in line with the philosophy of component-based systems, wherein specific components are replaced at runtime, depending on the current context. Social variability goes beyond this traditional system view. It considers open systems, where participants are autonomous and heterogeneous, and the overall system is an emergent concept that originates from the actions of individual participants and from their interactions.

3.4.1 Sociality and interaction

Social variability is about the existence and evolution of social relations. In this section we show that social relations originate from the interactions between actors. We justify this claim with the aid of literature in philosophy and multi-agent systems. This implies that most interactions between software systems—via exchange of messages—result in social relations between them. We limit our analysis to a particular kind of interaction, i.e. communication.

Let's start from the early work by J.L. Austin, whose book "How To Do Things With

Words" [Aus62] arose a revolution in philosophy of mind, and is the precursor of modern theories. In his work, Austin refuses the dominant theory at that time, which said that the purpose of sentences was to state facts that can become true or false. He provides solid examples against this theory, and introduces several kinds of sentences that are neither true or false. He names these sentences "performative utterances" (briefly, performatives); humans use performatives to carry out actions. Performatives are opposed to declarative or constative utterances—the true/false ones—that just describe the state of the world. An example of constative is "the sky is cloudy today", which is a statement about the state of the world that might be true or false. An example of performative is "I apologize for my behaviour", which is clearly not a description of the world.

Austin introduces three dimensions along which utterances can be analysed: *locutionary*, *illocutionary*, and *perlocutionary*. A locutionary act is the performance of an utterance, its ostensible meaning, that comprises phonetic, syntactic and semantic features. An illocutionary act refers to the use of a locution with a certain force such as stating, asking, commanding, promising. A perlocutionary act refers to the psychological consequences of a locution, such as persuading, convincing, scaring.

John Searle [Sea70, Sea75] refines the notion of illocutionary act introducing the term "speech act". Speech acts represent the semantic illocutionary force of an utterance, its intended meaning. Searle classifies speech acts [Sea75]: *representatives* (or assertives) commit a speaker to the truth of the expressed proposition ("today it's raining"), *directives* cause the hearer to take a particular action ("open that door!"), *commissives* commit the speaker to a future action ("I'll open that door tomorrow"), *expressives* communicate the speaker's attitude and emotion about the proposition ("congratulations for the award"), *declarations* change reality according to the proposition ("you are fired!").

The work by Austin and Searle shows how communication can be considered in terms of different perspectives, importantly its illocutionary force (the speech act). These theories had a great impact on multi-agent systems research. Mainstream agent-oriented programming languages, such as JADE [BPR99], exploit agent communication languages based on speech acts, in which the specification of messages includes the performative (the illocutionary act) the sender wants to communicate to the receiver.

As observed by Singh [Sin99], the concept of *social commitment* is useful to understand the classes of speech acts proposed by Searle. A social commitment is a four-place relation $\mathsf{C}(x, y, G, p)$ that denotes a commitment from agent x to agent y for the proposition p in the context of agent G. x is the debtor, y the creditor, G the context group, p the discharge condition. A conditional commitment is a commitment having a precondition: if the precondition holds, then the debtor commits to the creditor for the consequent p. Singh shows how commitments are integral part of speech acts.

Let's show some examples to illustrate how speech acts are representable as commitments. Commissives bring into effect a commitment by the speaker to the hearer. "I'll open the door tomorrow" means that the speaker commits to open the door tomorrow. Directives presume a commitment by the hearer to do as told. "Open that door!" presumes that the hearer will commit to open that door. Assertives commit the speaker to the statement expressed. "Today it's raining" is understood as a commitment from the speaker to the fact that it's actually raining. Notice that these are *social* commitments, and are different from psychological commitments [Sin91].

Our point here is that interaction—both in the physical and in virtual worlds—occurs in order to create, modify, and drop social relations. Specifically, we used existing literature to show that social commitments (just commitments, from now on) cover a wide range of illocutionary acts that are communicated via interaction. Figure 3.9 exemplifies the relation between interaction—via messages—and the corresponding social relations. The depicted scenario refers to a book-selling transaction between Bookie and Alice. Bookie sends an offer message to Alice saying that he will sell book BO if Alice pays \$12: this creates a commitment C_B from Bookie to Alice for BO. Then, Alice pays \$12; the meaning of this message is to inform Bookie that the payment has been made. Bookie is now unconditionally committed to bring about BO. Finally, Bookie sends a message saying that the book BO was given to the shipper for delivery; the meaning is that Bookie is informing Alice that the commitment proposition has been fulfilled. Consequently, the commitment is satisfied.

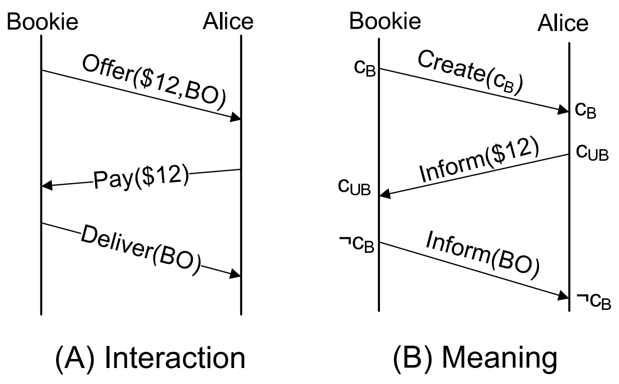


Figure 3.9: Interaction between two agents via messages (A) and its meaning (B)

Most system engineering approaches do not take into account the meaning of interaction between sub-systems. Our literature survey in Chapter 2 demonstrates this limitation and articulates how commitments are an adequate abstraction to overcome it.

3.4.2 Modelling service-oriented applications

In Section 3.4.1 we have shown that social relations are established via interactions between parties. We started from theories in linguistics and philosophy of mind, then we outlined how such theories apply to system engineering. In this section we propose a modelling framework for the design of systems that take into account social relations. The proposed framework is capable of representing and dealing with social variability.

Our modelling framework is designed for service-oriented applications. These applications exemplify the concept of programming-in-the-large introduced by DeRemer and Kron [DK76]. Programming-in-the-large suggests that large programs are not monolithic blocks, but are composed of a number of modules, possibly written by different people in different programming languages; these modules are linked together by an interconnection language that enables their interoperability. In a service-oriented application, the architecture of the overall application takes precedence over the specification of services. An individual service may be designed using any methodology in any programming language as long as it structurally fits in with the rest of the system. Component-based systems embody this philosophy, but service-oriented applications are fundamentally different in that they represent *open systems* [DNGM+08, SH05].

In our approach, a service-oriented application is characterized by the *autonomy* and *heterogeneity* of the participants. Application participants engage each other in a service enactment via interaction. Applications are *dynamic* implying that participants may join or leave as they please. The identity of the participants need not even be known when designing the application. Service-oriented applications take the idea of programming-in-the-large to its logical extreme. The ultimate purpose of this computational paradigm is to completely decouple the logic of a participant from the interaction that ties together the participants in the application.

An example of a service-oriented application are auctions on eBay. Multiple autonomous and heterogeneous participants are involved: the eBay corporation, buyers, sellers, payment processors, credit card companies, shippers, and so on. eBay (the organization) specified the architecture of the application in terms of the roles (seller, bidder, shipper, and so on) and the interaction among them without knowing the identity of the specific participants that would adopt those roles. Most likely, eBay was not aware it was building a service-oriented application while designing the eBay.com website. This example shows that this computational paradigm is already applied, though designers might be unaware of it. There is no question that this is an under-optimal situation, that can be improved by devising modelling frameworks that enable the systematic design of service-oriented applications.

It is widely recognized that, in (software) design, architectural models provide an adequate level of abstraction to represent the key structure and functionality of the designed artefact. For service-oriented applications, it is especially useful to treat the architecture as being largely synonymous with the application itself. In other words, there need not be any distinction between the concept of application and the abstraction that represents its architecture. From now on we will not differentiate between a service-oriented application and its architecture.

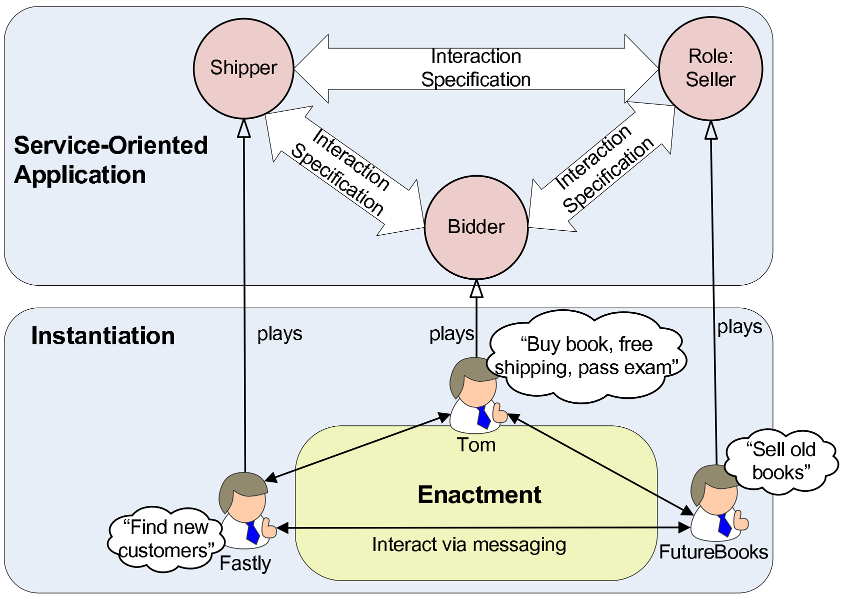


Figure 3.10: A service-oriented application is specified in terms of roles. It is instantiated when participants adopt those roles; it is enacted when participants interact according to the adopted roles

Service-oriented applications are characterized by some specific features that make them different from traditional software applications. We explain now these specificities with the aid of the eBay.com scenario. Such concepts are illustrated in Figure 3.10:

– A service-oriented application (from here on, simply application) exists irrespective of the existence of interacting participants. The auctions application on eBay exists whether some auction is going on or not;

– The application is specified in terms of roles. Actual participants are unknown at design-time. The eBay corporation doesn't know the actual agents that will create auctions or place bids;

– The application is *instantiated* when participants adopt (play) roles in the application. For instance, this happens as Bob signs in eBay.com using his *bob354* account and creates an auction for his old copy of “The Divine Comedy”. By doing that, he plays the role of “seller”;

– The application is *enacted* when participants interact according to the roles they play. For example, Bob interacts with a certain shipper to make sure his copy of “The Divine Comedy” is delivered to the winner of the auction.

– The application is, in general, specified independently from the specification of the individual participants. There are therefore at least two specification layers: that of the application—in terms of roles and their interaction—and that of the single participants. In line with the principles of programming-in-the-large, each participant can be specified in any language.

The real value of service-oriented computing is realized for applications in which participants engage each other in business transactions, such as auctions on eBay. Each individual participant has his own business goals, and he would need to interact flexibly with others so as to be able to fulfil his goals. Ideally, we would want to model both applications and participants in terms of business-level abstractions. We would also want to characterize and reason about properties critical to doing business, such as *goal fulfilment*, *compliance*, *interoperability*, and so on, in similarly high-level terms. This is key to alleviating the business-IT gap.

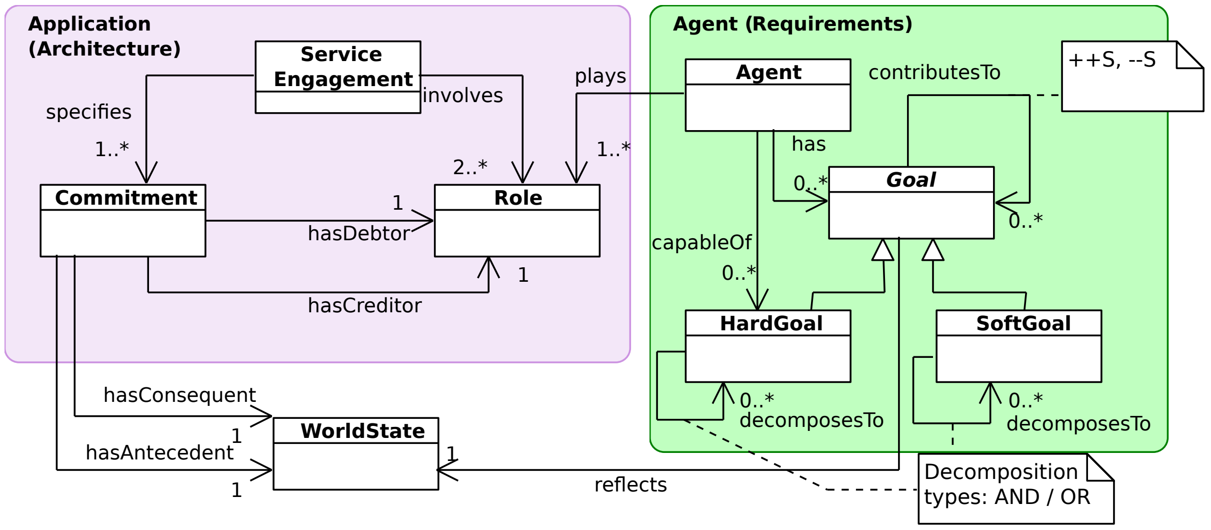


Figure 3.11: Conceptual model for service-oriented applications and participating agents

From here on, we refer to the participants in an application as *agents*. This is the most adequate term to accommodate their autonomy and heterogeneity. Notice that the concept of agent is different from its technological counterpart (software agents).

According to our notion, humans and organizations are agents too. Software agents act on behalf of organizations or humans. Figure 3.11 shows the proposed conceptual model: the left box concerns a service-oriented application (the STS), the right box is about an agent's requirements.

Our approach enables decoupling the specification of a service-oriented application from the specification of a specific agent. Each specification involves a different set of designers and stakeholders. In the following subsections, we will show how

In general, application designers specify the application without knowing who will be the actual agents participating in the application. Agent designers will specify an agent so that the requirements expressed by the stakeholders are satisfied. We do not provide detailed methodological guidance on how each artefact is to be engineered. However, the following subsections will describe how (i) a specific agent is specified using a goal model; (ii) a service-oriented application is specified in terms of a commitment protocol; (iii) an agent designer can check whether the agent's specification supports the agent's goals in a certain application.

Specifying agents via goal models

An agent is specified in terms of a goal model, as formalized in the meta-model of the Tropos methodology [BPG$^+$04]. As shown in the right side of Figure 3.11, an Agent *has* some goals. In other words, each agent aims to attain its current goals. A Goal may be a HardGoal or a SoftGoal. A soft-goal has no clear-cut criteria for satisfaction (its satisfaction is subjectively evaluated). A goal *reflects* a state of the world desired by the agent. In such a way, goals are grounded to something which is observable in the environment. A goal may contribute to other goals: $+\!\!+S(g, g')$ means that g contributes positively to the achievement of g'; $-\!-S(g, g')$ means that g contributes negatively to the achievement of g'. We do not consider here partial contributions between goals, but limit our framework to full positive or negative contributions. Both hard- and soft-goals may be AND-decomposed or OR-decomposed into sub-goals of the same type (hard goals are decomposed only to hard goals, soft-goals to soft-goals). Additionally, an agent may be capable of a number of hard-goals; the notion of capability abstracts the means-end relation in Tropos.

Specifying applications via service engagements

Our intent is to specify applications in terms of commitments. When specifying an application, commitments are expressed among roles. Later, when the application is instantiated and enacted, real agents play the roles in the specification. We use here the notion of conditional commitment, C(Debtor, Creditor, antecedent, consequent), in which the Debtor agent is committed to the Creditor agent for the consequent if the antecedent holds. The *antecedent* and the *consequent* are propositions that refer to the *states of the world* of relevance to the application under consideration.

Unlike the original formulation by Singh, we do not explicitly represent the context agent. We hypothesize it does not vary for all commitments within an application specification. A commitment is *discharged* when its consequent is achieved; it is *detached* when the antecedent holds. An unconditional commitment is one where the antecedent is $\top$ (true). For example, in an auction application, there might exist a commitment C(Bidder, Seller, bidWon, paymentMade). Its meaning is that the bidder commits to the seller that, if the world-state where the bidder has won the bid occurs, the bidder will bring about the payment.

We use commitments as the basis of architectural connections. As in Figure 3.11, a Service Engagement *involves* two or more roles and *specifies* one or more commitments among the involved roles. A Role role can be debtor (creditor) in one or more commitments; each commitment has exactly one debtor (creditor). A commitment has an antecedent and a consequent, each representing a specific state of the world.

Message	**From**	**To**	**Effect**	**Business Significance**
Create(x, y, r, u)	x	y	C(x, y, r, u)	brings about a relation
Cancel(x, y, r, u)	x	y	$\neg$C(x, y, r, u)	dissolves relation
Release(x, y, r, u)	y	x	$\neg$C(x, y, r, u)	dissolves relation
Delegate(x, y, z, r, u)	x	z	C(z, y, r, u)	delegates relation to another debtor
Assign(x, y, z, r, u)	y	x	C(x, z, r, u)	assigns relation to another creditor
Declare(x, y, p)	x	y	p	informs about some aspect of state

Table 3.1: Messages and their effects; a commitment is understood as a contractual relation

Commitments are dynamic relations. Indeed, agents manipulate commitments and make them evolve. Table 3.1 introduces the message types by which agents update commitments [CS09]. In the table, $x, y, \ldots$ are variables over agents, and $p, q, \ldots$ are variables over propositions. A Create message is sent by the debtor to notify the creditor about the establishment of a commitment. A Cancel message is sent from the debtor to dissolve a commitment; cancelling a commitment is a violation, for the debtor breaks his promise. A Release message is sent from the creditor to relieve the debtor from its commitment. A Delegate message is sent from the debtor to a third agent so that the third agent will commit to the creditor. An Assign message is sent from the creditor to the debtor so that the latter becomes committed to another creditor. A Declare message informs that a certain state of the world has been met.

Conceptually, a *service engagement* is a business-level specification of interaction. It describes the *possible* commitments that may arise between agents adopting the roles, and via the standard messages of Table 3.1, how the commitments are updated. Notice that such specification does not guarantee—by any means—that the agents will behave as prescribed. Due to their autonomy, agents can behave freely: they may either respect the specification or violate it. The context agent can deal with violations by applying new commitments. For instance, if you park your car in a forbidden area, police (the context agent) might assess a fine which makes you committed to pay a penalty. Similarly, in a service-oriented application like eBay.com, your account would be suspended or revoked by the eBay corporation if you don't behave according to the regulations.

Agent compliance amounts to the agent not violating any of his commitments towards others. A service engagement specified in terms of commitments does not dictate specific operationalizations (runtime enactments) in terms of when an agent should send or expect to receive particular messages; as long as the agent discharges his commitments, he can act as he pleases [DCS10].

Figure 3.12 shows the (partial) service engagement for an auction application. Figure 3.13 shows a possible enactment for the service engagement of Figure 3.12. The bidder first creates $\mathsf{C_B}$. Then he places bids, possibly increasing his bids (indicated by the dashed bidirectional arrow labelled "Bidding"). The seller informs the bidder that he has won the bid, which detaches $\mathsf{C_B}$ and causes commitment $\mathsf{C_{UB}} = \mathsf{C(Bidder, Seller, \top, paymentMade)}$ to hold. Finally, the bidder discharges his commitment by sending the payment.

$$\boxed{\begin{array}{l} \mathsf{C_B} = \mathsf{C(Bidder, Seller, wonBid, paymentMade)} \\ \mathsf{C_S} = \mathsf{C(Seller, Bidder, paymentMade, itemDelivered)} \\ \mathsf{C_A} = \mathsf{C(Auctioneer, Bidder, \top, itemCheckedForAuthenticity)} \end{array}}$$

Figure 3.12: A (partial) service engagement depicting an auction application. The labels are for reference purposes only. Figure 3.13 shows an enactment of this engagement between a bidder agent and a seller agent

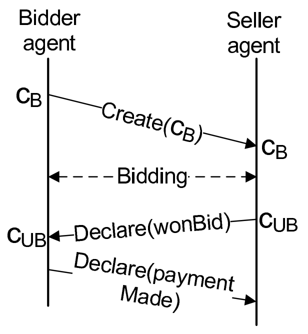


Figure 3.13: An enactment

A challenging research topic—that we do not address in this thesis—is the construction of service engagements. In general, domain experts specify service engagements from scratch or by reusing existing specifications that may be available in a repository. In eBay's case, presumably software architects, experts on the various kinds of businesses (such as payment processing, shipping, and so on) and processes (auctions) involved, and some initial set of stakeholders got together to define the architecture. How *application* *requirements* (as distinct from an individual participant's requirements) relate to the specification of service engagements is studied in the Amoeba methodology [DCS10].

Binding

As Figure 3.11 shows, an agent may choose to *play*, in other words, adopt one or more roles in a service engagement. Such an agent is termed an *engagement-bound agent*. Adopting a role is the key notion in instantiating an application, as shown in Figure 3.10.

However, before a bound agent may start interacting, he may want to verify that he is compatible with the engagement. The semantic relationship between a service engagement and an agent's goals is the following. To fulfil his goals, an agent would select a role in some service engagement and check whether adopting that role is compatible with the fulfilment of his goals. If it is compatible, then the agent would presumably act according to the role to fulfil his goals; else, he would look for another service engagement. For example, the bidder may want a complete refund from the seller if the seller delivers damaged goods. The bidder must check whether the service engagement with the seller includes a commitment from the seller to that effect; if not, he may opt for a different service engagement.

In Figure 3.11 both commitments and goals are expressed in terms of world states. This provides the common ontological basis for reasoning between goals and commitments.

3.4.3 The framework applied

We show how the modelling techniques can be used to represent two scenarios: the first one concerns car insurance claim processing, the second one is a simple version of an emergency response coordination STS focused on fire-fighting. Inspired by these scenarios, we apply the approach we proposed in Section 3.4.2, and we specify both the service-oriented application and one participating agent.

Claim processing: we base our scenario on the documentation that the Financial Services Commission of Ontario (FSCO) provides online, specifically on the description of the claim process[3]. The scenario describes the perspective of a driver involved in a car accident in Ontario, as well as highlighting what happens behind the scenes. The described engagement is independent of specific insurance companies, car repairers, and damage assessors. We assume the car driver is not at fault and his policy has no deductible.

Figure 3.14 describes the service engagement in the car insurance claim processing scenario. The engagement is defined as a set of roles (circles) connected via commitments; the commitments are labelled C_i. C_1 (insurer to repairer) states that if insurance has

[3] http://www.fsco.gov.on.ca/english/insurance/auto/after_auto_accident_ENG.pdf

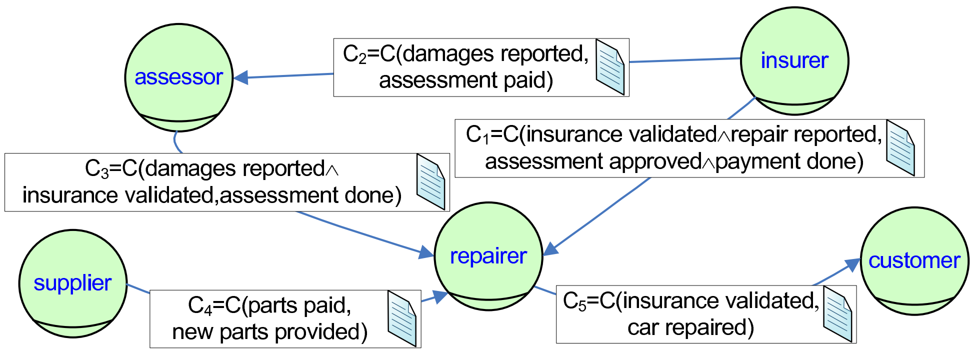

assessor
insurer
repairer
supplier
customer
C₂=C(damages reported, assessment paid)
C₁=C(insurance validated∧repair reported, assessment approved∧payment done)
C₃=C(damages reported∧ insurance validated,assessment done)
C₄=C(parts paid, new parts provided)
C₅=C(insurance validated, car repaired)

Figure 3.14: Role model for the insurance claim processing scenario. Commitments are rectangles that connect (via directed arrows) debtors to creditors

been validated and the repair has been reported, then the insurer will pay and approve the assessment. C_2 (insurer to assessor) says that if damages have been reported, the assessment will be paid by the insurer. C_3 (assessor to repairer) says that if damages have been reported and the insurance has been validated, a damage assessment will be performed. C_4 (supplier to repairer) says that if parts have been paid for, new parts will be provided. Finally, C_5 (repairer to customer) states that if the insurance has been validated, then the car will be repaired.

Figure 3.15 shows an agent model where agent Tony plays role repairer. An agent model is the specification of a single agent. The main goal of Tony is to perform a repair service. This is AND-decomposed to sub-goals car repaired, receipt sent, and service charged. The goal model contains two variation points: the OR-decompositions of goals parts evaluated and service charged. The former goal is OR-decomposed to sub-goals new parts provided and old parts fixed. Note the soft-goals low cost service and high quality parts. Using new parts has a negative contribution to low cost service and a positive one to high quality parts, whereas fixing old parts contributes oppositely to those soft-goals.

Fire-fighting. Jim is a fire chief. His root goal is to extinguish fires. Such goal may be achieved either by using a fire hydrant or a tanker truck. In order to achieve such goal, Jim exploits his own capabilities but also needs help from other agents. For instance, Jim needs an authorization to use fire hydrants.

Figure 3.16 shows the agent model of this agent. The goal fire extinguished is OR-decomposed to fire hydrant used and tanker truck used, representing the two basic ways to achieve the root goal. In order to use a fire hydrant, Jim needs to notify hydrant usage need and to get authorized. He is capable of notifying his need. In order to use a tanker truck, he has to pay the service, get the truck to the fire, and connect a water pipe. He is capable of connecting the water pipe to the truck.

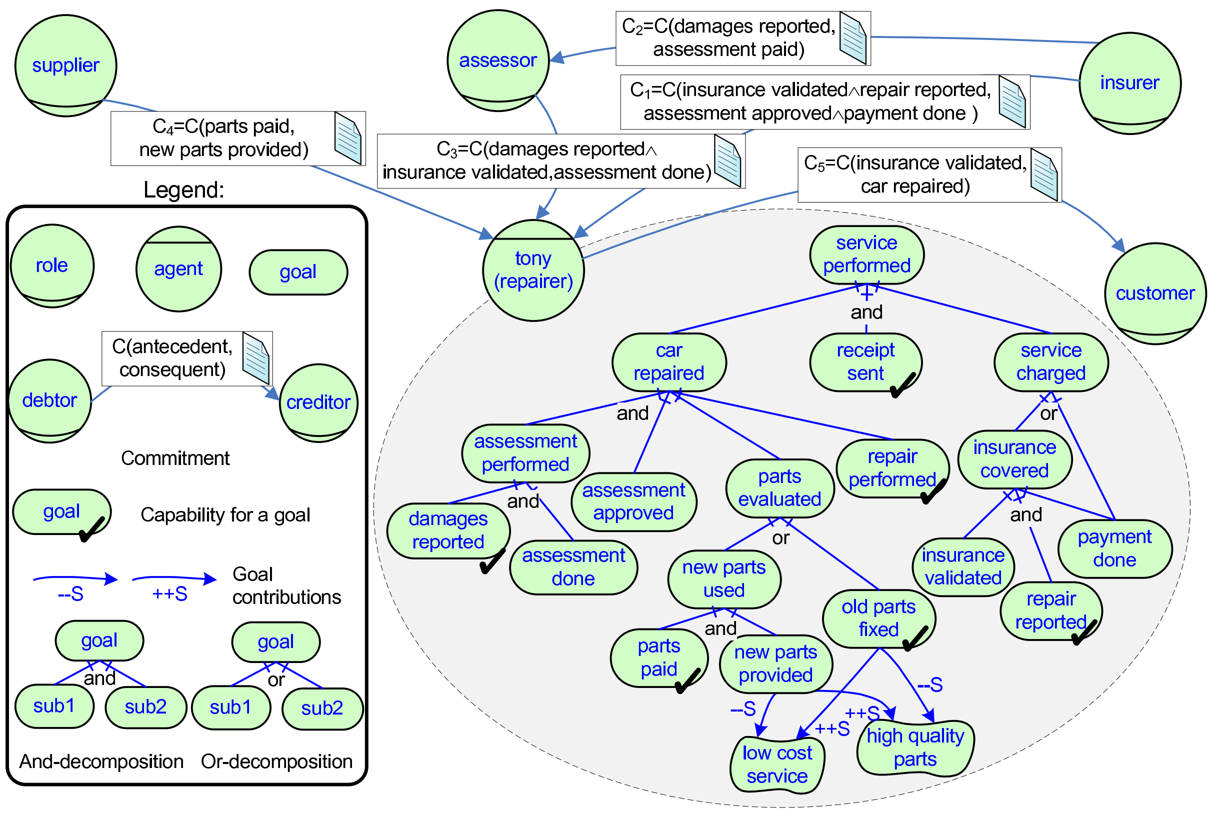


Figure 3.15: Visual modelling of Tony's engagement-bound specification. Tony plays repairer

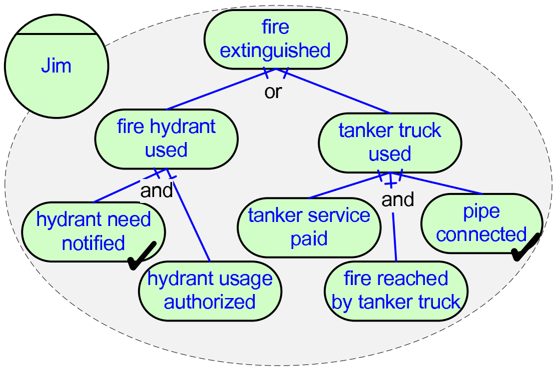


Figure 3.16: Agent model of Jim; he is a fire chief

Figure 3.17 shows a simple role model for the fire-fighting STS where Jim operates. A fire brigade commits to a fire chief that, if the hydrant need is notified, then the hydrant usage will be authorized (C_1). A fire chief commits to a fire brigade that, if the tanker service is paid, then the tanker truck will be used to put out a fire (C_2). A tanker provider commits to a fire chief that, if the tanker service is paid, then the fire will be reached by a tanker truck (C_3).

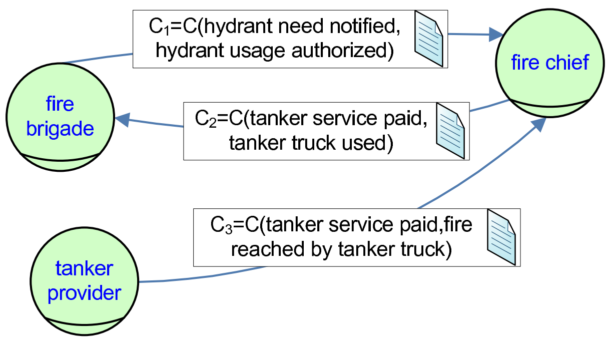

C_1=C(hydrant need notified, hydrant usage authorized)
fire chief
fire brigade
C_2=C(tanker service paid, tanker truck used)
C_3=C(tanker service paid,fire reached by tanker truck)
tanker provider

Figure 3.17: Simple role model for the fire-fighting example

3.4.4 Dealing with social variability

The modelling framework presented in Section 3.4.2 enables to represent: (i) service-oriented applications—via a role model that specifies the architecture in terms of the participating roles and the commitments between these roles—and (ii) the requirements of a specific participants—in terms of an agent model that specifies the goals of the participant and the relations between these goals.

We show now how such framework is able to represent the effect of social variability on requirements and enables automated reasoning about goal support. Three key features make our approach suitable for social variability:

- *The application is specified at the level of roles*: in open systems (exemplified by service-oriented applications), the actual participating agents are unknown at design-time and, also, they vary at runtime. The specification of the application is in terms of roles; in such a way, the architecture is a flexible one that is not bound to the participation of specific agents.

- *Actual commitments arise and evolve at runtime between agents*: the specification between roles does not prescribe any behaviour at runtime. It just represents the kind of commitments an agent can realistically expect in such application. Commitments arise at runtime as a consequence of the interaction between agents. Commitments evolve (e.g. they are discharged, cancelled, released, delegated) due to interaction between involved agents. They enable to represent the essential elements of social variability: the creation of commitments, the freedom of agents to join and leave, the manipulation of commitments.

- *Agents' rationale and social relations are at the same level of abstraction*: both the rationale of agents—expressed as goal models—and social relations—specified as commitments—are at the same level of abstraction and are ontologically linked to each other. This enables to link the private (an agent's internals) to the public (the interfaces expressed via commitments). We formalize such intuition through the notion of *variant* below.

We introduce the notion of a variant from the perspective of an agent. This formalization is not specific to any specific agent—it forms a common semantic substrate that characterizes a goal-oriented agent operating in a service-oriented application. The intuition is that, given a certain goal to achieve, a variant is a strategy that is likely to lead to the achievement of such goal—if the strategy is carried out successfully at runtime.

Let's first introduce the syntax of our language. Let $g, g', g'', g_1, g_2, \ldots$ be atomic propositions (atoms); $p, q, r, \ldots$ be generic propositions, either atomic or composite; $x, y, z, \ldots$ be roles. Let a_{id} be the agent under consideration. A *commitment* is a quaternary relation $\mathsf{C}(x, y, p, q)$. It represents a promise from a debtor agent playing role x to the creditor agent playing role y for the consequent q if the antecedent p holds. Let $\mathcal{P}$ be a set of commitments.

Commitments can be compared according to a basic strength relation [CS09]. If an agent commits for something, it will also commit for something less. Similarly, it will commit if he gets more than expected in return. This intuition is captured via the transitive closure of $\mathcal{P}$.

Definition 3.13 (Transitive closure) *Given a set of commitments $\mathcal{P}$, $\mathcal{P}^*$ is its transitive closure with respect to the commitments strength relation [CS09].*

Let $\mathcal{P} = \{\mathsf{C}(\text{fireman, brigade, team assigned} \lor \text{ambulance sent, fire fought} \land \text{casualties rescued})\}$. Then, for instance, $\mathsf{C}(\text{fireman, brigade, team assigned} \lor \text{ambulance sent, fire fought}) \in \mathcal{P}^*$, $\mathsf{C}(\text{fireman, brigade, team assigned} \lor \text{ambulance sent, casualties rescued}) \in \mathcal{P}^*$, $\mathsf{C}(\text{fireman, brigade, team assigned, fire fought}) \in \mathcal{P}^*$, $\mathsf{C}(\text{fireman, brigade, ambulance sent, casualties rescued}) \in \mathcal{P}^*$. Conversely, $\mathsf{C}(\text{fireman, brigade, team assigned} \lor \text{ambulance sent, casualties rescued} \land \text{traffic rerouted}) \notin \mathcal{P}^*$, $\mathsf{C}(\text{fireman, brigade}, \top, \text{casualties rescued}) \notin \mathcal{P}^*$.

Definition 3.14 (Goal Model) *A* goal model $\mathcal{M}_{id}$ *specifies an agent a_{id} as:*

1. *a set of AND/OR trees whose nodes are labeled with atoms;*
2. *a binary relation on atoms p-contrib;*
3. *a binary relation on atoms n-contrib.*

An AND/OR tree encodes the agent's knowledge about how to achieve the root node. The nodes are the agent's goals. *p-contrib*(g, g') represents positive contribution: the achievement of g also achieves g'. *n-contrib*(g, g') is negative contribution: the achievement of g denies the achievement of g'. Figure 3.16 is a goal model for agent Jim. $\mathcal{M}_{Jim}$ has one AND/OR tree rooted by goal fire extinguished. $\mathcal{M}_{Jim}$ contains no contributions.

We introduce the predicate *scoped* to capture a well-formedness intuition: a goal cannot be instantiated unless its parent is, and if a goal's parent is and-decomposed, all the siblings of such a goal must also be instantiated. The notion of *scoped* is useful to devise a clear definition of variant that does not consider partial states where an agent is transiting from one variant to another. The clarity and elegance we gain slightly sacrifices the expressive power of our framework, which can deal only with steady states.

Definition 3.15 (Scoped) *A set of goals $\mathcal{G}$ is* scoped *with respect to goal model $\mathcal{M}_{id}$, that is, scoped$(\mathcal{G}, \mathcal{M}_{id})$ if and only if, for all $g_0 \in \mathcal{G}$, either*

1. *g_0 is a root goal in $\mathcal{M}_{id}$, or*
2. *exists a simple path $\langle g_0, g_1, \ldots, g_n \rangle$ in $\mathcal{M}_{id}$ such that g_n is a root goal in $\mathcal{M}_{id}$ and $\forall i, 0 \leq i \leq n$:*
   (a) $g_i \in \mathcal{G}$, and
   (b) if anddec(g_{i+1}) $(i \neq n)$, then $\forall g$ such that parent(g_{i+1}, g), $g \in \mathcal{G}$

Example 1 $\mathcal{G}_1 = \{$fire extinguished, fire hydrant used$\}$ is well-formed with respect to $\mathcal{M}_{Jim}$. Indeed, fire extinguished is a root goal, whereas fire hydrant used is part of path $\langle$fire hydrant used, fire extinguished$\rangle$.

Example 2 $\mathcal{G}_2 = \{$fire extinguished, fire hydrant used, hydrant need notified$\}$ is not scoped with respect to $\mathcal{M}_{Jim}$. Though $\langle$hydrant need notified, fire hydrant used, fire extinguished$\rangle$ is a simple path to the root, the sibling of hydrant need notified—goal hydrant usage authorized—is not in $\mathcal{G}_2$. The set of goals $\mathcal{G}_3 = \{$fire extinguished, fire hydrant used, hydrant need notified, hydrant usage authorized$\}$ is scoped with respect to $\mathcal{M}_{Jim}$.

A variant is an abstract agent strategy for the achievement of some goal. It consists of a set of goals $\mathcal{G}$ that the agent intends to achieve via a set of commitments $\mathcal{P}$ and a set of capabilities $\mathcal{C}$. A variant is defined with respect to an agent's goal model $\mathcal{M}_{id}$. A variant is not a concrete strategy because it does not tell the exact course of action the agent should carry out to achieve its goals: which actions to execute, which messages to exchange, how to interleave actions, and so on.

Definition 3.16 (Variant for a Goal) *A triple $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor$ is a variant for a goal g with respect to goal model $\mathcal{M}_{id}$, that is, $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} g$ if and only if*

1. *scoped*$(\mathcal{G}, \mathcal{M}_{id})$ *and* $g \in \mathcal{G}$*, and*
2. g *is supported:* $\nexists g' \in \mathcal{G}$: *n-contrib*$(g', g) \in \mathcal{M}_{id}$*, and either*
   (a) $g \in \mathcal{C}$*, or*
   (b) $\mathsf{C}(x, a_{id}, g', g) \in \mathcal{P}^* : \lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} g'$*, or*
   (c) $\mathsf{C}(x, y, g, g') \in \mathcal{P}^*$*, or*
   (d) *ordec*(g)*, and either*
      i. $\exists g' : parent(g, g')$ *and* $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} g'$*, or*
      ii. $\mathsf{C}(x, a_{id}, g', q) \in \mathcal{P}^* : q \vdash \bigvee_{parent(g,g_i)} g_i$ *and* $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} g'$*, or*
      iii. $\mathsf{C}(x, y, p, g') \in \mathcal{P}^* : p \vdash \bigvee_{parent(g,g_i)} g_i$*;*
   (e) *anddec*(g) *and* $\forall g' : parent(g, g')$ *and* $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} g'$*, or*
   (f) *p-contrib*$(g', g) \in \mathcal{M}_{id} : \lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} g'$*.*

The goals $\mathcal{G}$ that a_{id} intends to achieve should be scoped with respect to the goal model $\mathcal{M}_{id}$ (clause 1). Goal g must be supported: there should be no negative contributions to g from any goal in $\mathcal{G}$ and one clause among 2a-2f should hold (clause 2).

2a. capabilities support goals;

2b. a_{id} gets a commitment for g from some other agent playing x if a_{id} supports the antecedent;

2c. some agent playing y brings about g in order to get a commitment for g' from some other agent playing x (possibly a_{id} itself);

2d. an OR-decomposed goal g is supported if either: there is some sub-goal g' such that $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} g'$ (2(d)i), g is supported via commitment to (2(d)iii) or from (2(d)ii) other agents. These two clauses cover the case of an agent who commits for a proposition that logically implies the disjunction of all the goal children. For instance, a commitment for $g_1 \vee g_2$ supports a goal g OR-decomposed to $g_1 \vee g_2 \vee g_3$;

2e. an AND-decomposed goal is supported if $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor$ is a variant for each of its children;

2f. positive contribution from g' supports g if $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} g'$.

Definition 3.17 generalizes the notion of variant to sets of goals. A variant for a goal set should be a variant for each goal in the set.

Definition 3.17 (Variant for a Goal Set) *A triple* $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor$ *is a variant for a goal set* $\mathcal{G}'$ *with respect to goal model* $\mathcal{M}_{id}$*, that is* $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} \mathcal{G}$*, if and only if, for all* g *in* $\mathcal{G}'$*,* $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{id}} g$*.*

We show now some examples to illustrate how the definitions of variant can be applied. We will verify whether a triple $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor$ is a variant for the root goal in $\mathcal{M}_{Jim}$ (Figure 3.16).

Example 3 Let $\mathcal{G} = \{$fire extinguished, fire hydrant used, hydrant need notified, hydrant usage authorized$\}$, $\mathcal{P} = \{\mathsf{C}_x = \mathsf{C}(x,$ Jim, hydrant need notified, hydrant usage authorized$)\}$, $\mathcal{C} = \{$hydrant need notified$\}$. $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{Jim}}$fire extinguished.

Step 1. From Definition 3.16, $\mathcal{G}$ must be scoped and fire extinguished supported.
Step 2. $\mathcal{G}$ is scoped: $\mathcal{G} = \mathcal{G}_3$ and $\mathcal{G}_3$ is scoped with respect to $\mathcal{M}_{Jim}$ (Example 2).
Step 3. Clause 2(d)i applies to fire extinguished if $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{Jim}}$fire hydrant used.
Step 4. Clause 2e applies to fire hydrant used if both $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{Jim}}$hydrant need notified and $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{Jim}}$hydrant usage authorized.
Step 5. $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{Jim}}$hydrant need notified holds by 2a: such goal is in $\mathcal{C}$.
Step 6. $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{Jim}}$hydrant need notified holds by Clause 2b: some other agent will commit for C_x to Jim, given that $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{Jim}}$hydrant need notified.

Example 4 Let $\mathcal{G} = \{$fire extinguished, tanker truck used, tanker service paid, fire reached by tanker truck, pipe connected$\}$, $\mathcal{P} = \{\mathsf{C}_y = \mathsf{C}($Jim, y, tanker service paid, fire extinguished$), \mathsf{C}_z = \mathsf{C}(z,$ Jim, tanker service paid, fire reached by tanker truck$)\}$, $\mathcal{C} = \{$pipe connected$\}$. $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{Jim}}$fire extinguished.

Step 1. From Definition 3.15, $\mathcal{G}$ is scoped with respect to $\mathcal{M}_{Jim}$. Indeed, fire extinguished is a root goal, tanker truck used is AND-decomposed, all its sub-goals are in $\mathcal{G}$, and there is a path from all goals in $\mathcal{G}$ to fire extinguished.
Step 2. From Definition 3.16, we should check if g is supported. Clause 2(d)i applies to fire extinguished if $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor \models_{\mathcal{M}_{Jim}}$tanker truck used.
Step 3. tanker truck used is AND-decomposed into tanker service paid, fire reached by tanker truck, pipe connected; 2e tells to verify every sub-goal.
Step 4. pipe connected is in $\mathcal{C}$, therefore 2a applies.
Step 5. tanker service paid can be supported if Jim commits for C_y to some agent (2c).
Step 6. fire reached by tanker truck is supported if some agent commits for C_z to Jim (2b), given that the antecedent tanker service paid is supported.

3.5 Chapter summary

We have investigated the role of software variability for software self-adaptation in STSs. In Section 3.1 we analysed the term *variability* showing its interpretations in different disciplines. In Section 3.2 we argued why variability is a fundamental concept for self-adaptive software, and we defined the core terminology of this thesis. In Section 3.3 and Section 3.4 we explored contextual and social variability, respectively. For each type of variability, we defined its meaning, described its impact, and proposed modelling primitives.

The main contribution of the chapter consists of the theoretical underpinning for our approach. In particular, the chapter proposes:

– The core terminology for the thesis: software variability, variant, adaptation, self-adaptation, and self-adaptive software. We have shown the generality of these definitions by applying them to different types of adaptive software.

– A modelling framework for contextual variability. First, we have characterized the notion of context and have explored its impact on requirements. Then, we have proposed the contextual goal model, which enables to systematically represent the relation between context and requirements. Contextual goal models represent the impact of contextual variability via contextual variation points.

– A modelling framework for social variability. Social variability is a main factor in STSs, where new systems—either technical or social—join and leave as they please, and the services they provide vary over time. After analysing social variability and describing how it is grounded in interaction, we have proposed and formalized a modelling for social variability based on the concepts of goal and commitment.

Chapter 4

An architecture for self-adaptive software

In this chapter we propose an architecture for self-adaptive software. The architecture applies to settings composed of multiple interacting actors, and is expressly thought for socio-technical systems. The purpose of the architecture is to ensure that an actor participating in the system achieves its goals. The architecture is conceptual: it describes the basic components that its implementations have to develop and outlines its runtime operation, but does not prescribe specific algorithms or technologies.

In Section 4.1 we present the principles for our conceptual architecture. In Section 4.2 we propose its logical view. In Section 4.3 we describe possible requirements models the architecture can use at runtime. Finally, in Section 4.4 we provide an account on how the architecture can be built for an existing system.

Acknowledgement. Section 4.2 is based on [DGM09a].

4.1 Underlying principles

We outline a set of basic principles for our architecture. They derive mainly from the characteristics of the setting the architecture is meant for, socio-technical systems. These principles guide the construction of the architecture, and ensure its adequacy to provide adaptation in STSs. Each principle is exemplified on the emergency response setting.

Principle 1 (Adaptation support) The architecture supports an agent in achieving its objectives by adding self-adaptation capabilities. □

The architecture provides self-adaptation capabilities to an agent. As stated by Definition 3.3, self-adaptation is performed to achieve the agent's objectives. An agent can be a human, an organization, or a software system. This principle is in line with externalized adaptation [GS02]: the adaptation logic is separated from the application logic.

Example 5 The architecture is installed on the PDA of a fire chief to support him in coordinating fire emergences response. The fire chief is responsible for defining and enacting a response plan. The architecture continuously monitors such plan and performs adaptations when the plan is at risk or is under-performing.

Principle 2 (Open socio-technical systems) The architecture operates in STSs composed of a varying set of interacting agents. □

This principle settles down the kind of setting the architecture is meant for. Being thought for settings characterized by openness, the architecture should effectively deal with their volatility. Two major dimensions of such volatility are contextual and social variability (see Chapter 3). Agents can join and leave as they please, they can offer new services or revoke existing ones, contextual entities and devices fail, the physical context is subject to expected and unexpected changes.

Example 6 During emergency response, multiple agents interact (fire chiefs, firemen, ambulances, doctors, police). They are free to join and leave: new firemen become available, doctors might exit to deal with other emergencies, each agent offers different services (e.g. doctor D_1 commits to rescue a patient in 10 minutes, doctor D_2 in 15 minutes). The severity of the fire varies, traffic jams might arise, weather conditions evolve. The architecture monitors these changes to detect threats and identify opportunities, so that the supported agent—the fire chief—can successfully achieve his purpose.

Principle 3 (Autonomy) The agents in the STS are autonomous. Therefore, the architecture cannot enforce their behaviour. □

Agent autonomy means that no agent (including our architecture) can force another to do any action. The architecture should foresee methods to bring about adaptations without forcing agents to behave in a certain way or another. For instance, the architecture might suggest what to do, send reminders, actuate changes using effectors in the context, or establish social relations with other agents. Not even the supported agent is controllable; however, it is presumably more receptive to adaptation suggestions than other agents.

Example 7 If a fire-fighter is under-performing—e.g. it is not using an adequate extinguishing agent to put out fire—the architecture might suggest to use a different tool or tell the fire chief to request an additional fire-fighter. However, the autonomy of each fire-fighter makes the adaptation strategy uncertain.

Principle 4 (Heterogeneity) The agents in the STS are heterogeneously constructed. □

Heterogeneity means that different agents are constructed differently. Together with Principle 3), this ensures that agents do not know the internal construction of other agents. In addition to the impossibility for an agent to control another, these principles say that an agent does not even know how another agent will think or act. Our architecture should therefore devise adaptation strategies that do not violate this principle.

Example 8 The architecture is likely to know the typical construction of the fire chief, given that it is built to support its objectives. On the contrary, it does not know how fire-fighters will reason and act. Therefore it can rely only on the social relations that can be established, such as their commitments to put out fires.

Principle 5 (Driven by requirements models) The architecture should be model-driven: models are kept alive at runtime to represent correct and incorrect system behaviour. These models should represent the agent's requirements. □

Our architecture should keep requirements models alive at runtime and use them to determine whether an agent is performing as expected. Requirements models give primacy to the purpose of the agent over the (low-level) mechanisms that the agent enacts to achieve requirements. This way, the use of requirements models in adaptation control loops guarantees that adaptation is performed to meet the agent's purpose.

Founding adaptation upon requirements models has some limitations. First, faults cannot be always detected. For example, incorrect context monitoring—e.g. due to faulty sensors—cannot be captured, for monitored data will be considered correct by the architecture. To overcome this limitation, our approach should be complemented by others that enable faults detection. Second, requirements models are assumed to be correctly specified. If a certain variant is applicable in a specific context—according to the requirements models—the architecture will be choosing that variant even if in practice the variant does not work (it is actually inapplicable). To deal with such issue, requirements models have to be monitored and revised at runtime (they should evolve).

Example 9 One of the requirements of a fire chief is to efficiently respond to fires in his competence area. Different high-level strategies can be devised to meet this requirement, such as assembling a fire squad or delegating the emergency to another fire chief. The architecture would check monitored data against these models to determine whether the purpose of the fire chief is threatened.

Principle 6 (Conceptual level) The architecture should be conceptual and domain-independent. □

The advantage of a conceptual architecture is that application domain details do not affect its applicability. Implementations for specific STSs specialize the conceptual architecture by considering domain-dependent characteristics. The architecture should not prescribe the usage of specific algorithms to identify the best alternative variant, while it can provide guidelines, such as general factors variant selection algorithms may consider.

Principle 7 (Adaptation control loop) The architecture operates via an adaptation control loop. Such control loop consists of four phases: Monitor, Diagnose, Reconcile, and Compensate. □

The importance of control loops is widely recognized in the area of self-adaptive software [BMSG$^+$09]. Our architecture will be based on a control loop composed of four main stages: (i) monitoring events that occur in the surrounding physical and social context (i.e. both context changes and messages exchanged between agents); (ii) diagnosing monitored data against requirements models to identify failures and under-performance; (iii) devising an abstract strategy that reconciles actual behaviour with expected behaviour; and (iv) enacting such strategy through compensation actions.

Example 10 Suppose the architecture is supporting the fire chief during a fire accident. The architecture detects that the fire severity is increasing, and that a new fire-fighter is available in proximity of the fire (he sends an SMS notifying its availability). The current variant is inadequate to cope with the fire, due to the increasing fire severity. Reconciliation is performed by selecting a variant where the new fire-fighter is assigned to that emergency. The compensation action is to send him a message asking him to get to the fire as soon as possible.

The novelty of the proposed architecture is to exploit high-level models to represent the system's purpose (requirements models) and to consider the social relationships between the supported system and other systems—either technical or social—in the STS. The usage of goal models and social commitments—to represent requirements and interaction, respectively—makes our proposal very flexible. By focussing on the *purpose* of the system and the *meaning* of interaction, adaptation guarantees that the system meets its stakeholders' strategic interests (the purpose) and is compliant with the commitments with other systems.

4.2 Logical view

Figure 4.1 presents our conceptual architecture for self-adaptive software. The figure shows an UML 2.0 component diagram that represents architectural components and the connections between them. The architecture is proposed at the conceptual level in accordance with Principle 6. Throughout this section we will use the terms adaptation and reconfiguration interchangeably.

Our architecture is founded on the Monitor-Diagnose-Reconcile-Compensate (MDRC) control loop. Hence, it complies with Principle 7:

1. Monitoring collects data about the state of the environment and the agents participating in the system from a variety of sources;
2. Diagnosis interprets these data with respect to requirements models to determine if all is well. If not, the problem-at-hand is diagnosed;
3. Reconcile searches for a new variant that best deals with the problem-at-hand;
4. Compensation takes necessary steps to define and execute a plan that enacts the new variant.

Our architecture is designed for socio-technical systems, which are inherently decentralized, distributed and heterogeneous. Therefore, our architecture has to take into account the interaction between participating agents and functional components as well as the supervisory MDRC cycle. These interactions are supported through additional elements of the architecture:

– *Context sensors* are computational entities providing raw data about the environment where the system runs. In the smart-home scenario, context sensors can determine the current location of the patient, temperature and humidity levels in specific rooms, open/closed status of doors and windows, incoming/outgoing phone calls, the location of other actors (caregivers, doctors) within the apartment or elsewhere.

– *Agents* include all the actors who need to be monitored to ensure that they deliver on their obligations to the system (i.e. they respect their commitments). These may be patients living in smart-homes, firemen and medical doctors in crisis management settings, air traffic controllers in charge to manage the air space around the airport they work in. A special kind of agent is the supported agent. In accordance with Principle 1, the architecture supports its requirements. Agents are interfaced with the system through interface *System pushes* so that they can be sent directions, advice and reminders. For instance, a patient may be reminded to take her medicine

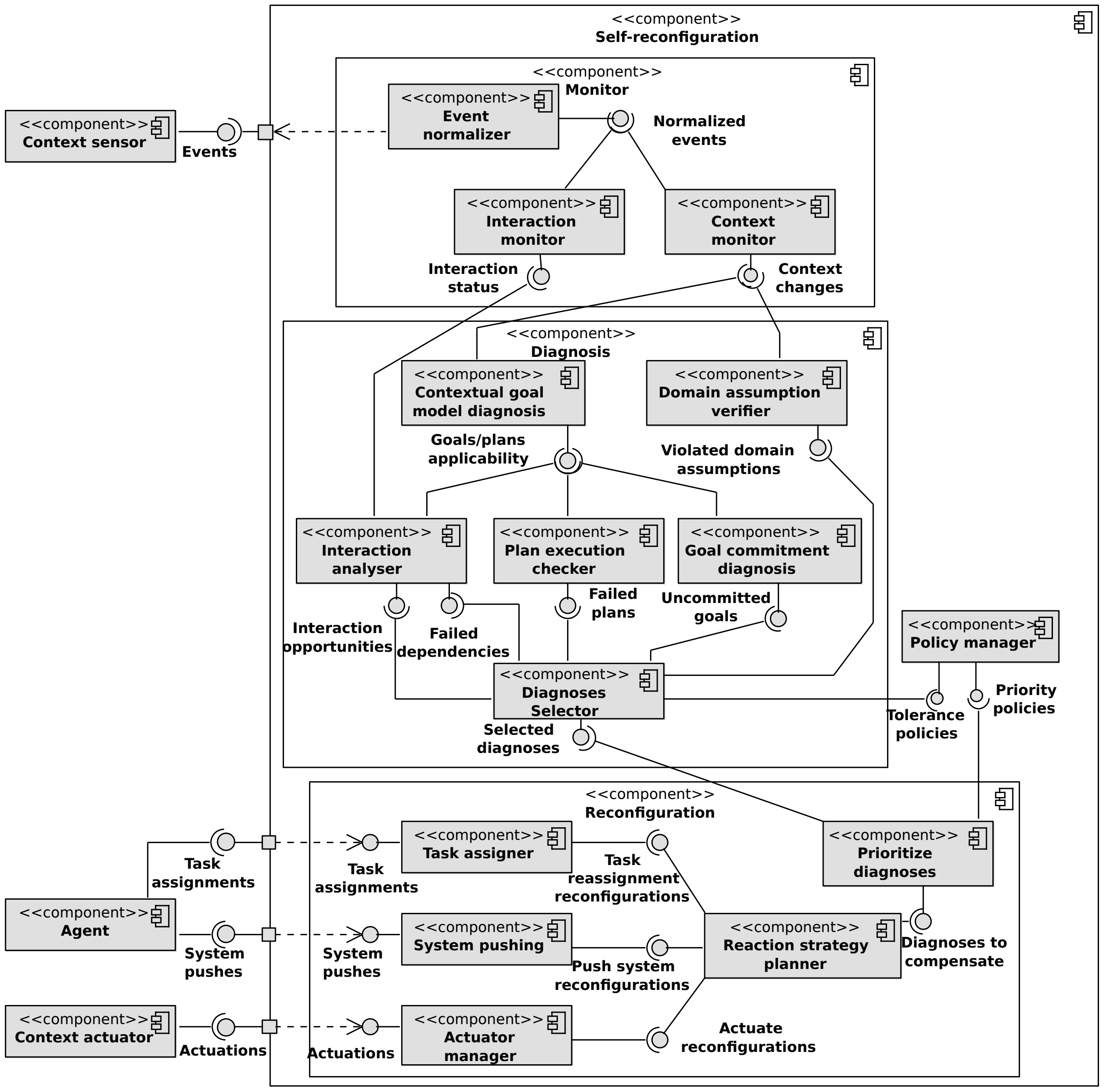


Figure 4.1: Logical view on the conceptual architecture for self-adaptation

by sending an SMS to her mobile phone. Also, the system can assign specific tasks to other agents—establish dependencies—through interface *Task assignments*. For instance, a catering service might be called to deliver food. These interfaces take into account the autonomy and uncontrollability of each agent (Principle 3).

– *Context actuators* represent any effector in the environment that can receive commands and act. Examples of actuators are sirens, door openers, automated windows, remote light switches, automatic 911 callers. The component receives the commands to enact through the required interface *Actuations*. Notice that agents and context actuators are mutually exclusive entities: while agents are autonomous, heterogeneous, and uncontrollable, context actuators are passive and controllable.

The *self-reconfiguration* component provides the self-adaptation capabilities of our architecture. This component is split into three sub-components. The *Monitor* component is in charge of collecting, filtering, and normalizing collected events; the *Diagnosis* component identifies symptoms and discovers root causes; the *Reconfiguration* component takes care of reconciliation and compensation to deal with such symptoms, which might be failures, under-performance, but also opportunities to better perform. These three components are detailed in the following subsections.

4.2.1 Monitor

The purpose of the *Monitor* component is to detect relevant changes in the physical and social context, and notify these changes to the diagnosis component. To collect events, this component relies on context sensors. Different sub-components are needed to carry out the function of the *Monitor*.

The *Event normalizer* component initiates the monitoring function, taking its input from the interface *Event* through appropriate ports (in Figure 4.1 ports are the small squares on component borders). The collected data is normalized to a common format that expresses the collected events on a context model (see Section 4.3.2 for an example of context model). Normalization requires the definition of a translation schema for each raw data format. If event sources provide data in standard formats (e.g., XML), transformation schemes can be defined using a transformation language (e.g., XSLT [C+99]). Figure 4.2 sketches how the event normalizer works. Three events are sent in different formats: binary raw data from the door, an XML file from the surveillance camera, CSV data from the thermometer. The event normalizer converts these data in terms of a shared context model.

The provided interface *Normalized events* is required by the components that deal with agent interaction and contextual events. The *Interaction monitor* computes the status of existing interactions and exposes it through interface *Interaction status*. For example, a social worker expected to visit might send a message to the the patient telling she cannot come. The *Context monitor* processes events related to context and exposes the interface *Context changes*. For instance, if the house entrance door is closed ($door.status = closed$) and an event such as $open(door, time_i)$ happens, the status of the entrance door will change to open ($door.status = open$).

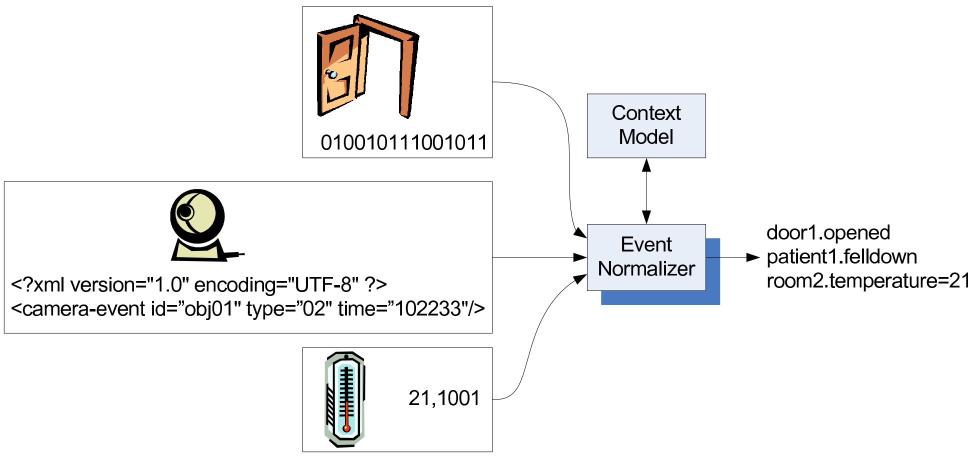


Figure 4.2: Overview of the *Event normalizer* component

4.2.2 Diagnosis

Diagnosis consists of checking current information about the system—collected by the Monitor component—against the requirements models kept alive at runtime.

The role of requirements models is to specify what should happen and hold: which goals should/can/cannot be achieved by certain agents, which plans they can/cannot execute in different contexts, the domain assumptions that should not be violated. On the one side, the richer the requirements models are, the more accurate the diagnosis will be. On the other side, the granularity of detected events is bounded by technological and pragmatic aspects. Detecting if a patient is sitting on a sofa is readily feasible (e.g. through pressure sensors), while detecting if she is handling a knife the wrong way is far more complex. Consequently, the level of detail of diagnosis is an application-specific parameter, and different models are (not) adequate depending on the specific application.

We will provide examples of possible specific requirements models in Section 4.3. Overall, the requirements models for the architecture should express (i) the goals of the supported agent in the STS; (ii) the context where the system is deployed in; (iii) the relation between goals and context; (iv) the interaction between the agents, in terms of social relations; (v) domain assumptions the system should not violate.

The *Contextual goal model diagnosis* component performs diagnosis on contextual goal models. It requires context changes, analyses contextual goal models to identify goals and plans that should/can/cannot be achieved, and provides this output through the interface *Goals/Plans applicability*. For example, it can detect that a patient is not having breakfast. Also, it can exclude variants involving preparing breakfast autonomously if the patient is sick in bed.

Domain assumption verifier also requires context changes, verifies the list of domain assumptions, and identifies violations that are then exposed through the interface *Violated domain assumptions*. Domain assumptions are contextual conditions that should not be violated in the system. For example, this component can verify requirements such as "The patient should never take his medicines twice after breakfast".

Interaction analyser requires interaction status and goals/plans applicability, and computes (i) failed dependencies between agents and (ii) interaction opportunities that have arisen. Dependencies fail not only if the dependee cannot achieve the goal or perform the plan (e.g., the nurse cannot support the patient because she's busy with another patient), but also changes in the context that make goals and dependencies inapplicable (e.g., the patient exits home and thus cannot depend on a catering service any more). New interaction opportunities arise if new agents join the system and offer some service—i.e. they commit to act as dependee—or they change their existing services.

Plan execution checker requires goals/plans applicability, determines failed plans and provides them through the interface *Failed plans*. This component verifies whether the way a plan is conducted is compliant with the plan specification. For example, it can identify failures such as "insulin was pumped while insulin pump was not under skin", or a timeout as the patient tries to wake up.

The *Goal commitment diagnosis* component diagnoses those goals that should be achieved for which the agent took no action so far. In other words, activated goals for which no plan has started yet. Notice that goal commitment refers to psychological commitments [Bra87], which is different from the social commitments we use to represent social interactions. For example, a goal commitment diagnosis might detect that the patient is not preparing breakfast. This component requires the interfaces goals/plans applicability and provides the interface *Uncommitted goals*.

Diagnoses selector requires failed dependencies, failed plans, uncommitted goals, interaction opportunities, as well as interface *Tolerance policies* provided by the *Policy manager*. The policy manager handles those policies defined by system administrators, e.g. cases where failures (or opportunities) do not lead to reconfiguration actions. For example, lack of goal commitment for washing dishes can be tolerated if the patient's vital signs are good (she may wash dishes after her next meal). Similarly, if a new catering service commits to provide breakfast but the patient is already depending on another provider, the new opportunity might be ignored. The interface *Selected diagnoses* contains the diagnoses that should be compensated via adaptation. Tolerance policies can be specified using an arbitrary language. For instance, Pimentel *et al.* [PSC10] extend our architecture with a policy language to deal with failures. Their language enables to express conditions such as "A task set can fail if a context holds", "A task set can fail if some goals are satisfied/unsatisfied", and "A task set can fail at most n times".

4.2.3 Reconfiguration

The reconfiguration phase defines compensation/adaptation strategies in response to selected diagnosis. The core idea is to devise a new variant that better achieves the goals of the supported agent.

The effectiveness of a reconfiguration depends on several factors: the number of plans that can be automated, the available compensation strategies, the effectiveness of suggestions and reminders on participating actors. The actual success of compensation strategies is scenario-specific and depends on available resources. Suppose a patient feels giddy: if she lives in a smart-home provided with a door opener, the door can be automatically opened to let the rescue team enter. Otherwise, the rescue team should wait for somebody—perhaps the porter—to bring the door keys.

The *Prioritize diagnoses* component requires selected diagnoses and priority policies, selects a subset of diagnoses according to their priority level, and provides them through the interface *Diagnoses to Compensate.* Priorities depend on failure severity, urgency of taking a countermeasure, time passed since diagnosis. For example, taking a remedy to failures is typically more urgent than exploiting new opportunities. The *Reaction strategy planner* component takes the diagnoses to compensate as input and selects a set of reactions to compensate for the failures. Given one or more failures, this component identifies possible reconfigurations, and selects one of them. Our architecture supports three types of reconfigurations:

– *Plan reassignment reconfigurations* involve the automated enactment of dependencies on external agents. The architecture acts on behalf of the supported agent to define a social relation with a dependee agent. A plan reassignment strategy works only if the dependee accepts to deliver the service (due to its autonomy, it is free to refuse). For example, if the patient has not had breakfast and the timeout for the goal is expired, the architecture can automatically call the catering service. If the catering service does not accept or later violates its task, the architecture will need to perform another reconfiguration.

– *Push system reconfigurations* remind agents their current goals or suggest plans to execute. Such option gives little certainty, but is often a very effective way when one deals with social actors. A push strategy for the patient that did not have breakfast so far is sending an SMS to her mobile phone.

– *Actuate reconfigurations* are enacted via context actuators. As we said earlier, context actuators are passive entities that can be commanded. For instance, if the patient feels bad, the door can be automatically opened by activating the door opener. In such a way, rescuers or neighbours can easily enter and help the patient.

Each reconfiguration type feeds a specific component (*Task assigner*, *System pushing* and *Actuator manager*) that enacts the chosen reconfiguration by interacting with external components (agents and actuators).

4.3 Requirements models for the architecture

As stated in Principle 5, our architecture is model-based and relies on requirements models. The logical view presented in Section 4.2 is in line with such principle: requirements models play an essential role during the entire adaptation control loop. Specifically, they are used to diagnose failures and under-performance, as well as in the reconciliation phase to identify new variants.

In this section, we present possible requirements models that can be used in implementations of our architecture. The logical view provides some guidelines about the type of models (contextual goal models, domain assumptions, interaction models, plan models), but does not mandate specific models. The models presented here are based on the modelling primitives defined in Section 3.

4.3.1 Contextual goal models

Contextual goal models enable to keep track of the supported agent's goals, as well as to identify alternative variants to achieve these goals. Our architecture supports any goal model that expresses the relation between goals and context. Our baseline is the contextual goal model we introduced in Section 3.3.3. We enrich it with information to enable its usage in the adaptation control loop of our architecture. We illustrate these changes with the aid of Figure 4.3. This goal model chunk refers to a patient living in a smart-home. The extensions we introduce are the following:

– *Activation rules* are associated to top-level goals. An activation rule is composed of a triggering event and a precondition. The top-level goal is activated when the triggering event happens, if the precondition holds. Active goals are those our architecture should monitor to detect threats and failures. Activation rules specialize of the "root goal" variation point described in Section 3.3.6. Figure 4.3 includes an activation rule for the top-level goal of the patient: goal Have lunch is activated when it's noon (triggering event), provided that the patient had not lunch before (precondition).

– *Time limits* are associated to top-level goals to define the maximum amount of time within which an agent has to achieve a goal. We assume that all models in the architecture represent time in terms of discrete time steps; the duration of a time

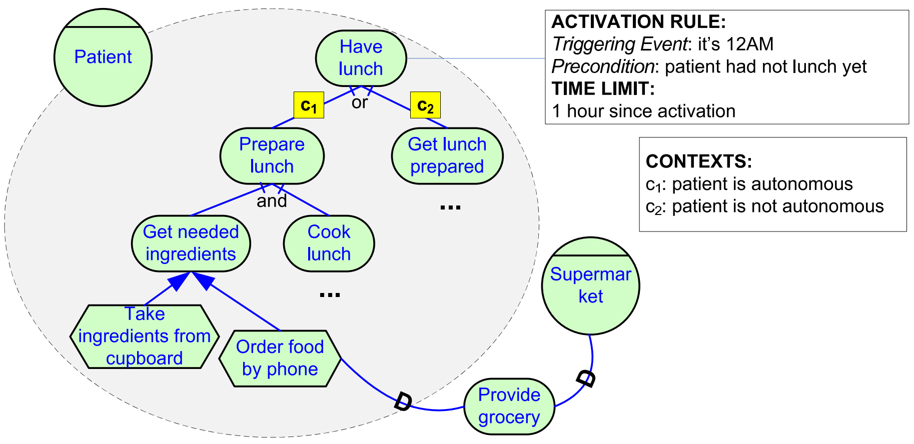


Figure 4.3: Contextual goal model with runtime extensions

step is domain-specific. In Figure 4.3, goal Have lunch should be achieved within one hour since its activation.

– *Plans* are sets of actions. In a contextual goal model, plans are connected to goals by means-end decompositions: an agent executes a plan to achieve a goal. To support plan monitoring and diagnosis, we specify plans using a simple and flexible language. Each action is performed correctly if its postcondition is achieved within a time limit and, at that time, the associated precondition holds. If an action is not performed correctly, the plan including it fails. We provide more details concerning plan specification in Subsection 4.3.3.

4.3.2 Context model

The context descriptions associated to variation points in contextual goal models define the impact of context on goals and variability. As we have explained in Section 3.3.3, contexts are typically expressed by analysts as abstract statements. These statements cannot be verified in an objective manner. Thus, our architecture cannot verify statements.

In Section 3.3.4, we proposed context analysis as a refinement process that reduces abstract contexts (e.g. "the patient is sick") to propositional formulae of observable facts (e.g. "the patient's oxymeter shows a saturation level below x" $\wedge$ "the patient did not wake up this morning"). Facts are expressed with respect to a contextual data model, which describes context in terms of entities, attributes and relations. The architecture monitors these facts on the contextual data model via contextual sensors.

We present here a context model based on class diagrams. Classes represent contextual entities (e.g. patient, oxymeter, home, smart-shirt, thermometer, etc.), attributes characterize the state of specific class instances (e.g. the heart rate of Bob's oxymeter, the temperature Jim's thermometer indicates), and associations define relations between different objects (e.g. patient Jim lives in smart-home #2, the thermometer t_x belongs to Jim).

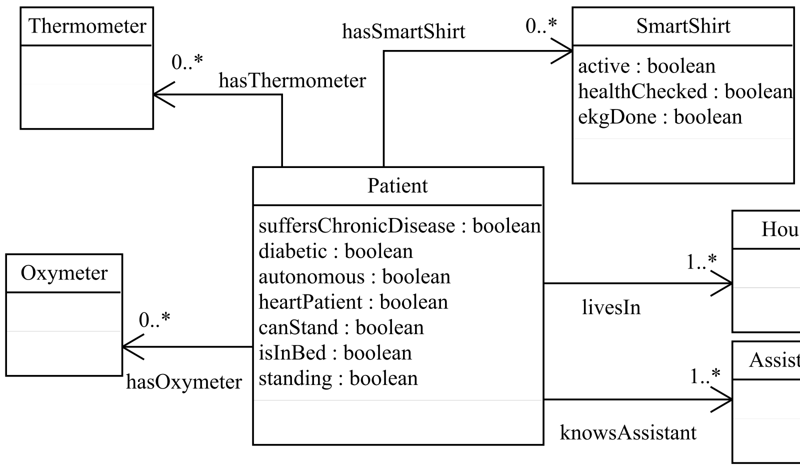


Figure 4.4: Part of the context model for the smart-home scenario

Figure 4.4 shows a piece of the context model for the smart-home scenario. The class "Patient" is characterized by a set of boolean attributes that express whether he suffers of a chronic disease, he is diabetic, autonomous, is a heart patient, can stand, is currently in his bed, and is currently standing. A patient has also associations to other classes: for instance, he can have zero or more smart shirts (association *hasSmartShirt* to class "SmartShirt"), and he knows at least one assistant (association *knowsAssistant* to class "Assistant").

At runtime, our architecture deals with instances of the classes in the context model. For instance, we might have two instances of patient (*jim* and *bob*), both living in the same smart-home *smartHome1*. Each of them might have one smart shirt (*ss1* and *ss2*). They might be assisted by the same assistant *mike*.

4.3.3 Plan specification

Goals are the basic concept we use to characterize the purpose of the supported system. They are abstract concepts, that agents achieve by executing concrete plans. There are many plan specification languages in literature: PDDL [FL03] is widely used in AI planning, JAM [Hub99] or AgentSpeak [Rao96] plans are used in BDI architectures, etc. These languages enable the specification of very detailed plans. In this section, we define a simpler plan specification language that allows for flexible plan execution (i.e. actions are not sequential unless explicitly specified).

We specify a plan as a set of actions to be carried out. Each action is characterized in terms of (i) a *postcondition*, the expected effect produced by performing that action; (ii) a set of *preconditions*, the state of the world that should hold to enable an agent perform that action; (iii) a *time limit* within which the action should be carried out. Preconditions can be critical or non-critical. If an action postcondition is met and a critical precondition does not hold, the action leads to plan failure.

We formalize the semantics of plans in the statechart in Figure 4.5. The statechart represents the possible states for a plan and the transitions between these states. The meaning of each state is the following:

– *idle*: None of the actions in the specification has been executed so far, the time limit is not expired. This state includes situations in which some actions started, and none of them is done yet.

– *started*: at least one (but not all) action has been performed successfully, the time limit is not expired. This state represents the non-atomic nature of plans: several actions have to be performed before completing the plan, and in that situation the plan is in progress;

– *success*: all actions in the specification have been executed successfully within the time limit. This is a terminal state: once a plan succeeds, there is no need for the system to monitor its execution;

– *failed*: at least one action in the specification has been performed, and at least one critical precondition was false. This is another terminal state. Plan failure triggers an adaptation process, unless the application policies tell to ignore such failure;

– *timeout*: the time limit expired. Timeout is a particular type of failure, and is a terminal state.

To better explain our plan specification language and clarify how it can be used to determine the status of a plan, we illustrate it on the smart-home scenario. Suppose the patient's goal is to get up, and the smart-home system has to ensure he achieves such objective. A possible way for the patient to get up is to grab a bed pole. So, plan get up by using pole is specified by actions A1 and A2 as follows:

A1. *Precondition*: patient p is in bed b, bed b has a pole pl; *Postcondition*: the bed pole pl is touched; *Time limit*: 5 minutes;

A2. *Precondition*: patient p is in bed b (critical), bed b has pole pl, pl is touched; *Postcondition*: patient p stands up; *Time limit*: 10 minutes;

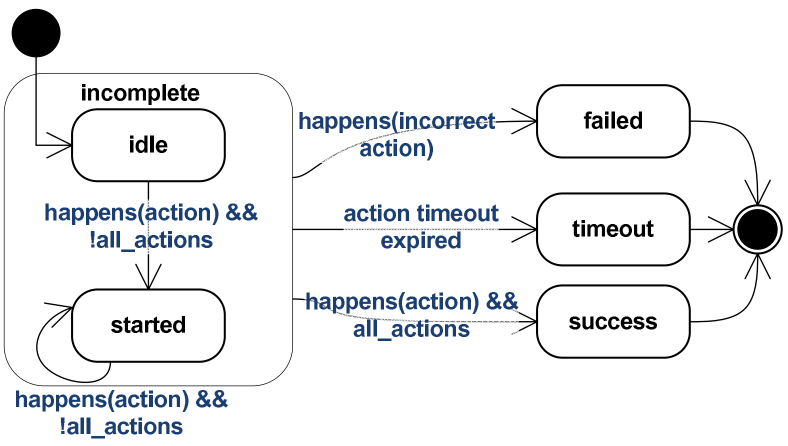


Figure 4.5: Statechart showing the possible state transitions for a plan

The correct execution of plan get up by using pole requires that: (i) the bed pole is touched within five minutes since plan activation (the time the means-end decomposed goal is activated), and (ii) the patient stands up—after he touches the bed pole—within ten minutes. There are three possible violations for this plan:

– the patient stands up without being in bed before. This might represent a fault in the camera that detects when the patient stands up, or in the bed sensor that detects if the patient is in bed;

– a timeout failure happens if the patient doesn't stand up within ten minutes (e.g. since the alarm rings). This happens if the patient touches the bed pole—his intention is to stand up—and he does not stand up within the time limit;

– the pole is not touched within five minutes. Both this violation and the previous one indicate a possible health problem of the patient.

Based on the policies defined by designers, these failures might result in different actions: the system might notify a nurse and give her access to the room webcam, gently alert the patient, or ignore the failure if his vital signs are good.

Consider now another goal for Frank, who is a heart patient: check vital signs. Such goal is triggered after meals. A possible plan is to use an oxymeter. Suppose such plan is defined by two actions: "the oxymeter measures the heart rate" and "the oxymeter measures the saturation level". Both actions have the same preconditions: "the patient has an oxymeter" and "such oxymeter is working". The latter precondition is critical: if the oxymeter measures the heart rate (or the saturation level) but it is not turned on, then something is wrong in that plan. Perhaps the on/off sensor is not working, or there is a major fault in the oxymeter, and maybe patient has not actually measured his vital signs. The time limit for these actions is 16 minutes: if 16 minutes pass since the alarm ringed, the plan is marked as timeout (failure).

4.3.4 Domain assumptions

A domain assumption is an indicative property that should not be violated by the system or its surrounding environment [JMF08]. Domain assumptions are a common type of requirements. Though they do not refer to functions the system should provide, they should be considered during design, since they express stakeholders' needs or constraints. Our architecture supports monitoring and diagnosis of domain assumptions.

We specify domain assumptions as implications over the context model (e.g. that introduced in Section 4.3.2). The implication antecedent consists of an activation event, that triggers the domain assumption, and an arbitrary number of preconditions. If the antecedent holds (the activation event happens while the preconditions are true), the consequent has to be verified. We allow for two types of consequent: (i) a state should hold when the activation event happens; and (ii) an event should happen within a time limit since the antecedent event happens.

A possible domain assumption for the smart-home scenario is "The oven should be turned off within 10 minutes if the patient is not at home". This domain assumption is represented as follows: (i) the activation event is "the patient exits home"; (ii) the antecedent precondition is "the oven is turned on"; (iii) the consequent is event "the oven is turned off", which should occur within ten minutes. Another domain assumption is that "the fridge door should be closed if the patient is not in kitchen". This can be expressed as follows: (i) the activation event is "the patient exits kitchen"; (ii) the consequent is state "the fridge is closed".

4.3.5 Interaction modelling via commitments

A socio-technical systems is a multi-agent system. Consequently, requirements models should be able to represent the interaction between agents. The social relations that arise from interaction should be identified and their status verified. As explained in Chapter 2.1, commitments are an effective abstraction to capture the meaning of interaction and keep track of the active social relationships.

As described in Section 3.4.1, the socio-legal context where interactions occur defines a mapping between domain-specific messages—exchanged between agents—and their meaning in terms of commitments. So, for instance, an "offer-lunch(10)" message sent from a catering service to the patient means that a commitment $\mathsf{C}(\mathsf{catering}, \mathsf{patient}, \mathsf{tenEurosPaid}, \mathsf{lunchDelivered})$ is created. Similarly, a message such as "lunch-not-needed-anymore" from the patient to a catering service means that the catering is released from its previous commitment. Again, a message "john-will-deliver-lunch" from the catering to the patient represents a delegation from the catering service to John for lunch delivery.

Given this mapping between domain-specific messages and their meaning, the system should keep track of the evolution of the commitments. Figure 4.6 illustrates how different message types—corresponding to commitment manipulation operations—modify the state of a commitment. We assume that commitments have an expiration deadline and the debtor should bring about them within a deadline. A commitment is initially in a

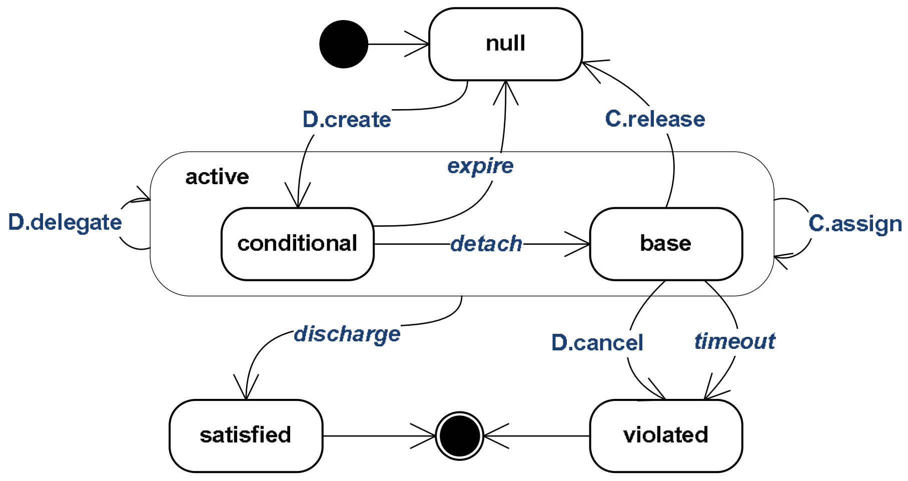


Figure 4.6: Statechart representing the state evolution of commitments (adapted from [SCD09])

null state. Then, the debtor creates it and makes it active (in particular, *conditional*). If the antecedent is brought about, the commitment is detached and the debtor is unconditionally committed (*base*). Commitments typically have an expiration deadline; once it expires, commitments are dropped and go back to a *null* state. If a creditor releases a debtor from a base commitment, the commitment state goes to *null*. The debtor can delegate active commitments; similarly, the creditor can assign active commitments. If the consequent of an active commitment is brought about, the commitment is discharged and goes to state *satisfied*. A commitment can be discharged even if the antecedent is not brought about. Base commitments can be cancelled by the debtor or expire by timeout. In both cases, the commitment is *violated*.

We illustrate these concepts on an example. Suppose a catering service $\mathsf{Cat1}$ sends a message to patient Jim committing to deliver lunch and a snack within 2 pm if a 20 euros payment is made within 11 am. This creates a conditional commitment $\mathsf{C}_1 = \mathsf{C}(\mathsf{Cat1}, \mathsf{Jim}, \mathsf{twentyEurosPaid}, \mathsf{lunchDelivered} \wedge \mathsf{snackDelivered})$. If Jim makes no payment within 11 am, then C_1 expires. Alternatively, if Jim pays within 11 am, commitment C_1

is detached and base commitment $\mathsf{C}_{1'} = \mathsf{C}(\mathsf{Cat1}, \mathsf{Jim}, \top, \mathsf{lunchDelivered} \wedge \mathsf{snackDelivered})$ holds. Suppose now Cat1 delivers lunch at 1 pm: base commitment $\mathsf{C}_{1''} = \mathsf{C}(\mathsf{Cat1}, \mathsf{Jim}, \top, \mathsf{snackDelivered})$ holds. If Cat1 delivers the snack within 2 pm, then the commitment is discharged (and satisfied). If Cat1 tells it won't be able to deliver or does not deliver on time, the commitment is violated.

4.4 Applying the architecture

To exploit the proposed architecture, requirements engineers and software designers need guidance to apply it to existing socio-technical systems. Once implemented, the architecture can be deployed to add self-reconfiguration capabilities to an STS by helping the supported agent achieve its objectives. We propose here a methodological process to apply our architecture. Figure 4.7 shows it as a SPEM 2.0 [Obj08] diagram represent-

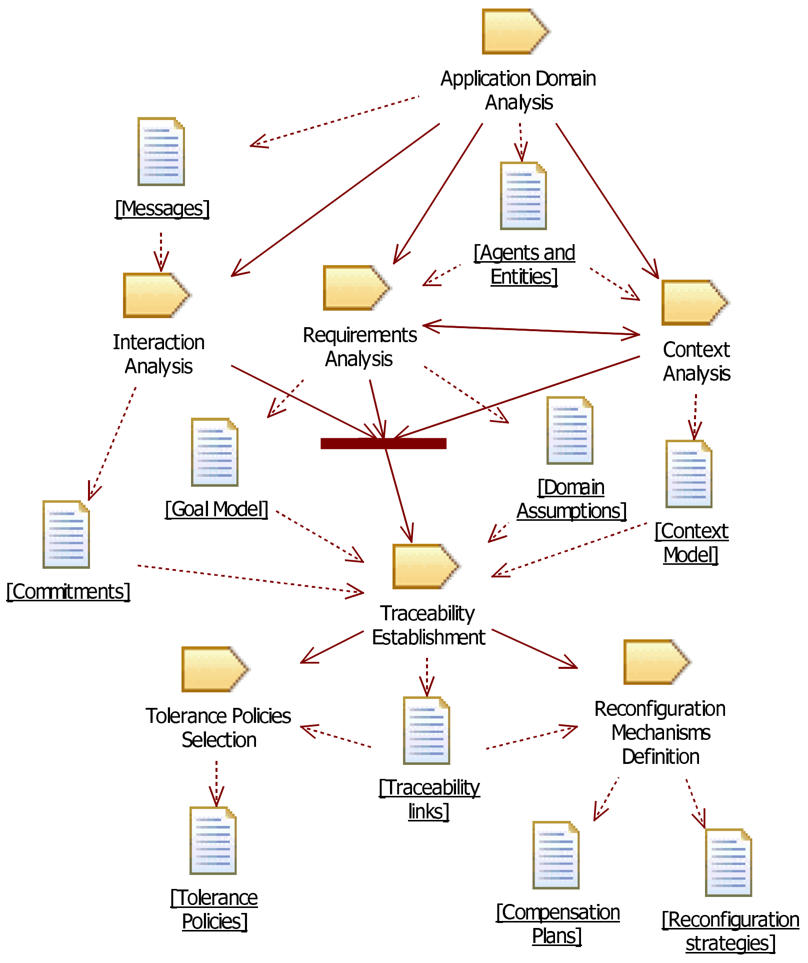


Figure 4.7: SPEM 2.0 diagram showing how to create the architecture for an STS

ing the sequence of activities—continuous lines—and the input-output flow in terms of artefacts—dashed lines. The diagram provides high-level guidance to successfully apply the architecture. However, specific methodologies might be devised to refine and better specify such process.

The first task is *Application Domain Analysis*, which corresponds to acquiring knowledge about the STS in terms of humans, organizations, and autonomous software agents as well as non-autonomous entities such as sensors and actuators. The resulting artefact is the set of agents and entities in the application. Also, such activity provides a list of domain-specific messages that are exchanged between participants in the STS.

Three tasks should be concurrently performed after domain analysis: *Requirements Analysis*, *Context Analysis*, and *Interaction analysis*. The output of these activities consists of the models we presented in Section 2.1: context analysis produces a context model; requirements analysis generates goal models and domain assumptions; interaction analysis defines commitments.

Context and requirements analysis are typically performed iteratively: the corresponding models are not isolated. Indeed, requirements models include links to contextual information. For example, domain assumptions are implication over context entities. Also, the status of goals (active, started, done, failed) is characterized as contextual events and states. Given the tight connection between these models, changes in the context (requirements) model often requires to modify the requirements (context) model.

After requirements and context analysis are completed, *Traceability Establishment* is performed. This task defines what to monitor at runtime in order to determine requirements satisfaction and violation. The objective of this step is to ensure traceability links between implementation and requirements. In particular, requirements reflection [Fin08, BWS$^+$10] should be established to enable a system to be aware of its requirements. This step is carried out by associating a specification—in terms of monitorable actions expressed over the context model—to goal model tasks. Traceability for commitments means defining the mapping between domain-specific messages and their meaning in terms of commitment operations. Communication via commitments might be enabled by an API that keeps track of commitments status and enables commitments manipulation.

After establishing traceability, the next steps are to select tolerance policies and to define reconfiguration mechanisms. Task *Tolerance Policies Selection* specifies tolerance policies for failures and under-performance. Some failures have to be addressed through reconfiguration, some can always be tolerated, some can be tolerated under certain circumstances. Policy definition is first enacted at design-time, but in most cases policies need to be redefined or adjusted at runtime. For instance, system administrators might realize that too many failures are considered, and the system is spending too much time in generating or enacting new variants.

Task *Reconfiguration Mechanisms Selection* defines how failures should be addressed by the architecture. The activity produces two output artefacts: (i) *Compensation Plans* are intended to revert the effects of the failed strategies, and (ii) *Reconfiguration Strategies* describe a possible alternative to achieve current goals. Both steps depend on the actuation capabilities of the existing application: possible reactions are scenario-specific, since the architecture needs to control actuators or communicate with specific agents.

4.5 Chapter summary

In this chapter we have presented a conceptual architecture to structure self-adaptive software for STSs. In Section 4.1 we presented the principles for our architecture. In Section 4.2 we described the logical view on the architecture. In Section 4.3 we described the requirements models the architecture exploits; they are based on the modelling frameworks to deal with variability we presented in Chapter 3. In Section 4.4 we provided methodological guidance to support engineers in applying our architecture.

The main contribution of this chapter lies in the conceptual architecture we have proposed. It has been devised after carefully identifying a set of basic principles: the focus on adaptivity in open STSs, the autonomy and heterogeneity of subsystems, adaptation based on a model-driven approach, and the usage of requirements-level abstractions. The control loop of the architecture is based on a Monitor-Diagnose-Reconcile-Compensate (MDRC) cycle: events in the environment are monitored; failures are diagnosed by checking events against requirements models; a strategy to reconcile actual behaviour with expected one is generated; finally, compensation actions are taken to enact the strategy.

The architecture supports the objectives of one agent. It enacts adaptation taking into account the autonomy of participants in an STS. Compensation actions cannot force other agents to enact plans (agents cannot be controlled); however, the architecture can remind agents of their commitments or suggest them an alternative course of action. Other compensation actions are controlling some actuator in the environment (unlike agents, actuators are controllable) and negotiating the assignment of plans with other agents.

In addition, we have proposed possible requirements models that the architecture can use at runtime. First, we refined the modelling languages for contextual and social variability introduced in Chapter 3 with runtime extensions. Second, we introduced simple languages to specify flexible plans and domain assumptions.

Chapter 5

Diagnosis and reconfiguration algorithms

In this chapter we detail a set of algorithms that can be used in our architecture. The algorithms we present apply to the requirements models introduced in Chapter 3 and refined in Section 4.3. These algorithms demonstrate how the architecture can operate to provide self-adaptation capabilities. Other algorithms can be devised to improve their performance or to support different requirements models. These algorithms have been implemented in the prototypes described in Chapter 6.

The chapter consists of three parts. Section 5.1 describes algorithms to perform goal, plan, and commitment diagnosis. Section 5.2 introduces algorithms to generate possible variants and to select the best variant. In particular, we present two algorithms based on soft-goals and cost, respectively. Finally, in Section 5.3 we introduce some adaptation tactics (patterns) that can be employed in STSs characterized by heavy social variability.

Acknowledgement. Section 5.1 is partially based on [DGM09a], while Section 5.3 extends [DCGM10].

5.1 Diagnosis algorithms

We describe diagnosis algorithms that check monitored events against the requirements models presented in Chapter 3. Monitored events (the current *behaviour* of the system) are compared to *expected behaviour*, which is expressed by the requirements models. A failure occurs when (i) the monitored behaviour is not allowed (e.g. a patient has to heat up her room, and she opens the window in winter); or (ii) expected behaviour does not occur (e.g. a catering service committed to deliver breakfast within half an hour does not deliver in time).

We split this section into three parts, each analysing a different kind of failure. Section 5.1.1 describes diagnosis for goal failure; Section 5.1.2 focuses on plan failure; Section 5.1.3 outlines how commitment violations are identified. We report on performance through application to case studies and scalability analysis later in the thesis (Chapter 7).

5.1.1 Goal failure

Since supported agents are specified via goal models, goal failures should be detected. Goals represent the business rationale of an agent. The goals of a software agent are assigned to it by its stakeholders. Consider, for instance, a software agent that supports a patient in a smart-home, e.g. it regularly checks the patient's health. Failing in checking her health corresponds to compromising the purpose of the system—providing prompt detection of and quick response to health problems of the monitored patient.

$$\frac{\begin{array}{l} goal(g,P) \ \wedge \ happened(activation\_evt(g,P),t) \\ \wedge \ \neg done(g,P) \ \wedge \ \nexists \ g_p, dec \ s.t. \ decomposed(g_p,g,dec) \end{array}}{\mathbf{should\_do(g,P)}} \qquad \text{(i)}$$

$$\frac{should\_do(g,P)}{\mathbf{visible(g,P)}} \qquad \text{(ii)}$$

$$\frac{\begin{array}{l} goal(g,P) \ \wedge \ \neg done(g,P) \\ \wedge \ \exists \ g_p \ s.t. \\ \quad goal(g_p,P_p) \ \wedge \ decomposed(g_p,g,dec) \\ \quad \wedge \ visible(g_p,P_p) \ \wedge \ holds(context\_cond(dec)) \\ \quad \wedge \ \forall p \in P \ s.t. \ (\exists \ p_p \in P_p \ s.t. \ name(p) = name(p_p)), \\ \qquad value(p) = value(p_p) \end{array}}{\mathbf{visible(g,P)}} \qquad \text{(iii)}$$

$$\frac{\begin{array}{l} plan(t,P) \ \wedge \ holds(pre\_cond(t,P)) \ \wedge \ \neg done(t,P) \\ \wedge \ \exists \ g \ s.t. \\ \quad goal(g,P_p) \ \wedge \ means\_end(g,t,dec) \\ \quad \wedge \ holds(context\_cond(dec)) \ \wedge \ visible(g,P_p) \\ \quad \wedge \ \forall p \in P \ s.t. \ (\exists \ p_p \in P_p \ s.t. \ name(p) = name(p_p)), \\ \qquad value(p) = value(p_p) \end{array}}{\mathbf{visible(t,P)}} \qquad \text{(iv)}$$

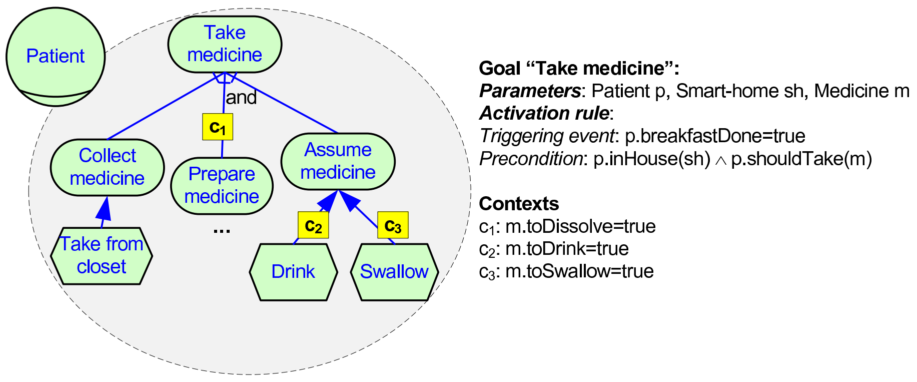
Table 5.1: Expected and visible goals and plans for an agent

To detect when goals fail we first need to identify which are the goals to achieve, and the sub-goals that can be achieved, in the current context. For this purpose, we introduce two predicates to denote goals that should be achieved (*should_do*) and goals/plans that can be achieved/executed (*visible*). The semantics for these predicates is given in Table 5.1

in first-order logic.

This formalization supports multiple instances for the same goal class. At design-time, goal models contain goal classes; at runtime, these classes are instantiated. Goals are parametric: their instances differ for the actual parameters that bind to the formal parameters. A patient's goal *Have breakfast* is repeated every day with different values for the parameter *day*, and it has different instances for each patient living in the same smart-home. Let's explain the rules in Table 5.1.

- Rule (i) defines when a top-level goal—i.e. a goal having no parent—should be achieved. This happens if g is a goal instance with actual parameters P, it has not been achieved so far, the activation event has occurred, and there exists no goal g_p that is AND/OR-decomposed into g.
- Rule (ii) is a general axiom saying that whenever a goal instance should be achieved, it is also visible (i.e., it is compatible with the current context).
- Rule (iii) defines when decomposed goals are visible. A goal instance g with parameter set P can be achieved if it has not been achieved so far and exists an achievable goal g_p with parameters P_p that is decomposed into g, the context condition on the decomposition holds, and the actual parameters of g are compatible with the actual parameters of g_p.
- Rule (iv) defines visibility for plans that are linked to goals via means-end decompositions. This rule is very similar ro rule (iii), with two main differences: plans are also characterized by a precondition—which should hold to make the plan executable—and are connected to goals through means-end.



Goal "Take medicine":
Parameters: Patient p, Smart-home sh, Medicine m
Activation rule:
Triggering event: p.breakfastDone=true
Precondition: p.inHouse(sh) $\wedge$ p.shouldTake(m)

Contexts
c_1: m.toDissolve=true
c_2: m.toDrink=true
c_3: m.toSwallow=true

Figure 5.1: An example from the smart-home scenario to show visibility

Figure 5.1 exemplifies the notion of visibility through an example from the smart-home scenario. The figure shows a top-level goal Take medicine, which is activated after the patient has breakfast. The same patient can have multiple instances of that goal at the same time, one for each medicine he should take. In order to take medicine, the patient collects the medicine, prepares the medicine, and assumes it. The medicine has to be prepared only if it should be dissolved in water (context c_1 holds). Suppose the patient has had breakfast, and thus he should take a medicine that is to be dissolved in water. In such context, goal Take medicine should be achieved, all sub-goals and tasks are visible, expect for Swallow. Indeed, the medicine is not to swallow. If the medicine were to swallow and not to dissolve in water, goal Prepare medicine and task Drink would not be visible.

Algorithm 5.1 Identification of goal and plan failures

```
DiagnoseGoal(GoalPlan  $g$)
 1  GoalPlan [] $failure, started, done, children$
 2  $g.status \leftarrow$ uncommitted
 3  if IsGoalDecomposed($g$)
 4    then $children \leftarrow$ GetSubgoals($g$)
 5         for each $g_i$ in $children$ do $\langle failure, started, done\rangle$ += DiagnoseGoal($g_i$)
 6         if $\forall g_i$ in $children$, $g_i.status$ = success
 7           then $g.status \leftarrow$ success
 8                $done$ += $\{g\}$
 9    else  $children \leftarrow$ GetMeansToEnd($g$)
10  if $g.status$ = uncommitted
11    then for each $g_i$ in $children$
12         do if !$g_i.visible$ then continue
13            if IsAndDecomposed($g$)
14              then if $g_i.status$ = fail
15                     then $g.status \leftarrow$ fail
16                          break
17                     else if $g_i.status$ = in_progress then $g.status \leftarrow$ in_progress
18              else if IsMeansEndDecomposed($g$) then $g_i.status \leftarrow$ CheckPlan($g_i$,$g$)
19                   switch $g_i.status$
20                     case success :
21                          $g.status \leftarrow$ success
22                          $done$ += $\{g\}$
23                          break
24                     case in_progress :
25                          $g.status \leftarrow$ in_progress
26                     case fail :
27                          $failure$ += $\{g_i\}$
28                          if $g.status$ = uncommitted then $g.status \leftarrow$ fail
29  if $g.status$ = in_progress then $started$ += $\{g\}$
30  if ($g.should\_do$ and $g.status$ = uncommitted and Timeout($g$)) then $g.status \leftarrow$ fail
31  if $g.status$ = fail then $failure$ += $\{g\}$
32  return $\langle failure, started, done\rangle$
```

Algorithm 5.1 (DiagnoseGoal) applies rules (i)-(iv) to diagnose goals and plans failures. DiagnoseGoal is invoked for each top-level goal instance the agent should achieve. Internally, it is recursively invoked to consider sub-goals and plans in the considered goal tree. The algorithm declares the arrays to contain failed, started and done goals/plans, also one array for the sub-goals or means-end decomposed tasks (*children*). Then (line 2) the status of goal g is set to `uncommitted`, since no information is initially available.

Lines 3-9 define the recursive structure of the algorithm. If the goal is AND/OR-decomposed (line 3), the array *children* is initialized to contain all the sub-goals of g (line 4), and the function DiagnoseGoal is recursively called for each sub-goal (line 5), updating the goal arrays. If the status of all the sub-goals is `success`, then the status of g is also set to `success`, and g is added to the array `done` (lines 6-8). If the goal is means-end decomposed (line 9), *children* is assigned to the set of plans that are means to achieve g.

If the status of g is still uncommitted (line 10), each sub-goal (or means-end decomposed plan) in *children* is examined (lines 11-28). If a children g_i is not visible, the algorithm does not examine the goal further and continues with the next element in *children*; indeed, invisible goals cannot be achieved in the current context. If g is AND-decomposed (lines 13-17), two cases are handled: if g_i failed, then the status of is g is set to *fail* and no other element in *children* has to be examined (lines 14-16); else if g_i is in progress, the status of g is set to `in_progress` (line 17).

If g is OR-decomposed or means-end decomposed (lines 18-28), it succeeds if at least one sub-goal (or plan) succeeds. If g is means-end decomposed, the algorithm calls CheckPlan (Algorithm 5.2 in Section 5.1.2) to diagnose plan status (line 18). If the status of g_i is `success`, the status of g is also set to `success`, g is added to the set of achieved goals, and the cycle is terminated (lines 20-23). If g_i is in progress, the status of g is set to `in_progress` (lines 24-25). If the status of g_i is `fail`, g_i is added to the set of failures, and, if g is still uncommitted, its status is set to `fail` (lines 26-28).

If the loop is over and g is `in_progress`, g is added to the set of started goals (line 29). If g is a top-level goal to achieve, its status is `uncommitted`, and the timeout expired, then the status of g is set to `fail` because the agent took no commitment for g (line 30). Notice that commitment here refers to a psychological commitment, and not to social commitments between agents. If the status of g is `fail`, it is added to the list of failures (line 31). The algorithm returns failed, started and succeeded goals (line 32).

We illustrate now how the algorithm works on the example in Figure 5.1. Suppose the status of task Take from closet is success, context c_1 is false, c_2 is true, and the status of task Drink is success. Then, the status of goals Collect medicine and Assume medicine is success (see line 7). Since the state of both visible sub-goals of Take medicine is success, the top-level goal is in state success too. Consider now a different scenario, where contexts c_1 and c_2 are true, the status of Take from closet, Prepare medicine, Drink are success, failed, and in progress, respectively. Here, the failure of Prepare medicine—which is now visible—is propagated to the top-level goal, whose status is therefore `failed` (see line 15). If context c_1 were false, the status of Take medicine would be in progress (see line 17), for the failure of Prepare medicine would have no effect, given that the goal is not visible.

5.1.2 Plan failure

A plan is a course of action that an agent carries out to achieve a goal. In our contextual goal modelling framework (Section 3.3.3), plans are linked to goals via means-end relations. In the commitment-based framework (Section 3.4.2), plans are abstracted—to emphasize the social aspects of the framework—by the concept of capability, but they exist and are performed by agents when they use their capabilities.

We refer here to the plan specification formalism we introduced in Section 4.3 and, specifically, to the statechart of Figure 4.5. A certain plan is idle (`uncommitted`) if no action in its specification has been performed; once actions in the specification are executed, the plan status can be in progress (`in_progress`) or failed (`fail`). Failure occurs if an action is executed and its critical precondition was not true. If all actions are executed correctly, the state of the plan turns to `success`. If an action time limit expires, the status turns to `timeout`, which is a special kind of failure.

Algorithm 5.2 Plan execution diagnosis

```
CheckPlan(GoalPlan means, GoalPlan end)
 1  int $start\_time \leftarrow$ GetActivationTime($end$)
 2  Event [ ] $events \leftarrow$ GetSpecification($means$)
 3  $plan\_status \leftarrow$ uncommitted
 4  for each $evt$ in $events$
 5  do if $\exists time1 > start\_time$ s.t. Happened($evt, time1$)
 6        then if $time1 > start\_time + evt.time\_limit$
 7                then return fail
 8             boolean $done \leftarrow$ true
 9             for each $prec$ in $evt.preconditions$
10             do if !HoldsAt($prec, time1$)
11                   then if IsCritical($prec$)
12                           then return fail
13                           else $done \leftarrow$ false
14             if $plan\_status$ = uncommitted and $done$
15                then $plan\_status \leftarrow$ success
16        else if $plan\_status$ = success
17                then $plan\_status \leftarrow$ in_progress
18  return $plan\_status$
```

Algorithm 5.2 (CheckPlan) describes diagnosis for plans. Its parameters are a plan and a goal linked by a means-end decomposition. Lines 1-3 initialize the variables used by the algorithm: *start_time* contains the activation time of the goal; *events* is the set of events that correspond to the execution of actions in the plan specification; and *plan_status* is the return value of the algorithm, initially set to `uncommitted`. All the events in the specification are examined (for cycle in lines 4-17). If the examined event has happened after the activation of the goal (lines 5-15), the event is further analysed. If the event happened after the event time limit, the algorithm returns failure (lines 6-7); otherwise, the event preconditions are examined (lines 8-13). Line 8 initializes the variable *done* to true. If the precondition does not hold (line 10) and it is critical (line 11) the algorithm returns failure (line 12), whereas if it is not critical the variable *done* is set to false. After preconditions are checked, if the plan is still uncommitted and the variable *done* is still true (line 14), the plan status is set to success (line 15). If the event has not happened after goal activation and the plan status is `success` (line 16), the plan status is set to `in_progress`, since not all events happened. Line 18 returns the plan status if the algorithm has not returned failures in the cycle.

Let's exemplify this algorithm with scenarios inspired by Figure 5.1:

– Consider plan Take medicine from closet, whose specification is as follows: (a) closet opened [time limit=4 minutes]; and (b) medicine removed, if (critical precondition) closet opened [time limit=6 minutes]. Suppose event "medicine removed" happens after 3 minutes, with the closet currently closed. In such case, the algorithm returns status "failure" in line 12, for the critical precondition for action (b) does not hold. Suppose now event "closet opened" happens one minute before "medicine removed". In such situation, the plan status is success, for all actions happen without violating critical preconditions and within time limits. An example of plan in progress is when the closet is opened at minute 3 and the medicine has not been removed yet.

– Take now plan Drink specified by action "glass emptied", with critical precondition "medicine dissolved", and time limit 3 minutes. Suppose that event "glass emptied" happens after four minutes; this leads to a timeout failure. Consider an alternative scenario wherein event "glass emptied" happens after two minutes, but the medicine has not been dissolved. Again, the action fails (status `failure`). If event "medicine dissolved" happened at minute 2, then the status would be `success`.

5.1.3 Commitment violation

Commitments are the abstraction our framework exploits to represent the interactions between different agents. From the perspective of one agent, commitments are the social relationships that it has made to or taken from other agents to achieve its current goals. An agent needs to identify promptly commitments violations, since they could threaten its current goals and, thus, require the agent to adapt.

We distinguish between two major types of commitment violation: explicit and implicit. Both violations are captured in the statechart for commitment evolution that we presented in Figure 4.6:

– An *explicit violation* occurs when a debtor cancels a commitment that he has made before. This cancellation is the effect of a message sent by the debtor to the creditor. This message is bound to commitment cancellation by the mapping between domain-specific messages and commitment operations. Such mapping exists in the socio-legal context where the commitment is enacted. Also notice that it is possible that the socio-legal context introduces penalties to debtors violating their commitments or other types of compensation actions (e.g. new commitments).

– An *implicit violation* occurs when an unconditional commitment is not achieved within the commitment time limit. Such violation is implicit because it is not originated by an exchange of messages. To detect implicit violations, there should exist a default time clock in the socio-legal context where commitments are enacted. Agents cannot question the validity of such time clock.

Let's exemplify these violations with the aid of the smart-home STS. Suppose a doctor Doc1 commits to the health service Health that, if the patient Jim in the smart-home has a fever, then the doctor will visit him within 24 hours. This is encoded as a commitment $\mathsf{C(Doc1, Health, jimHasFever, jimVisited)}$, with a 24 hours time limit. Such commitment can be violated in various circumstances. First, Doc1 might send a message to the health service saying that he is sick. The domain-specific interpretation of such message is a cancellation that does not imply penalties but defines a compensation action, the commitment of Doc1 to find another doctor who can visit Jim. Another explicit violation is Doc1 sending a message to the health service saying that he cannot arrive in time. In such case, a penalty might be applied. Implicit violations happen when the commitments time limits expire: Doc1 received a message saying that Jim is feverish, and 24 hours have passed since the receipt of such message.

5.2 Reconfiguration algorithms

We propose two algorithms to identify possible variants and to select one of them. These activities are fundamental part of any adaptive control loop, and output the variant to be enacted. These algorithms introduce general criteria that an agent can exploit to generate variants and to select the most promising one.

The two proposed algorithms focus on different aspects of variant generation and selection. The first algorithm (Section 5.2.1) focuses on the usage of soft-goals—that represent qualitative objectives—as a criteria di select the best plan to achieve the agent's current goals. The second algorithm (Section 5.2.2) exploits commitments to define variants that involve interaction with other agents, and chooses the variant that costs the least.

5.2.1 Soft-goal based

Our modelling frameworks represent correct behaviour of an agent in terms of a goal tree, where high-level goals are AND/OR-refined until concrete means (tasks) to achieve these goals are identified. AI planning and replanning algorithms are a good candidate to identify possible plans for goal achievement. There are three main planning mechanisms to identify alternative variants in goal models in response to a specific failure:

- *Backtracking*: the goal tree is explored bottom-up in order to find an alternative plan. The principle is that options in the same tree branch are preferable to options in other branches. Backtracking preserves stability—it ensures that the new variant differs as little as possible from the current one—but does not guarantee the optimization of cross-cutting concerns, such as soft-goals. We have experimented this approach in [DGM09b], where we apply this technique by extending the agent-oriented programming language Jason [BWH07]. Backtracking for failure handling is extensively discussed by Sardina and Padgham [SP07].

- *Tree planning*: this class of mechanisms corresponds to planning from scratch in a hierarchical task network. Tree planning identifies the best solution in the goal tree (based on the metric used to determine the quality of a solution), regardless of how much the new variant differs from the previous one.

- *Tree replanning*: this approach combines tree planning and variant stability. Algorithms in this category do not only consider plan optimality, but also minimize changes from the previous configuration. The impact of each of these two factors (optimality and stability) can be balanced depending on the agent's preferences.

The algorithm we present belongs to the *replanning* category. Plan optimality is determined via contribution to soft-goals, while stability considers the distance between the old and the new variant computed as the compensation cost for the plans that belong to the current variant and not to the new one. The reconfiguration process is enacted by Algorithms 5.3 and 5.4, which perform variant generation and selection, respectively.

Algorithm 5.3 Variants generation

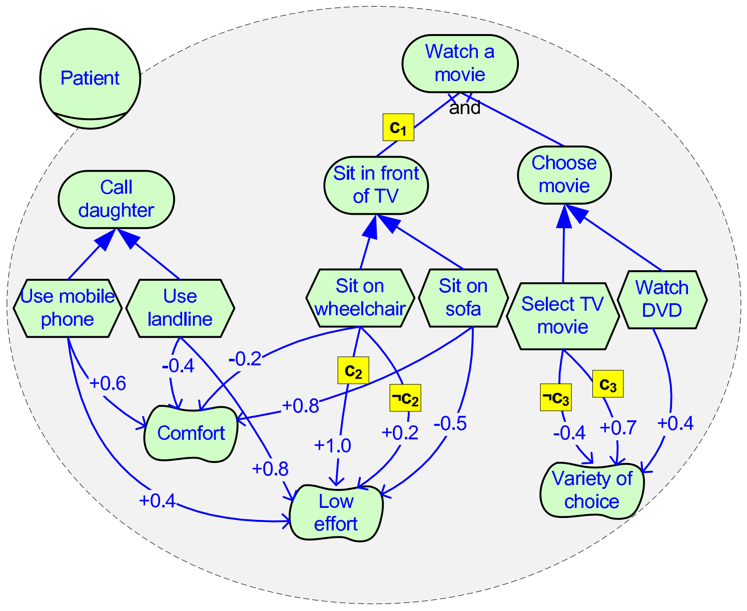
```
GenerateAlternatives(Agent $ag$)
 1  PlanSet [][] $options \leftarrow$ null
 2  for each $g \in$ GetShouldDo($ag$)
 3  do PlanSet [] $opts \leftarrow$ GeneratePlans($g$)
 4      $options$.Add($opts$)
 5  PlanSet [][] $cartProduct \leftarrow options[1] \times \ldots \times options[n]$
 6  Variant [] $variants \leftarrow$ null
 7  for $option \in cartProduct$
 8  do Variant $v$
 9      for $planSet \in option$
10      do $v$.AddPlanSet($planSet$)
11      $variants$.Add($v$)
12  return $variants$
```

Algorithm 5.3 takes as input the supported agent in the STS, and generates all alternative variants that satisfy the current goals of that agent. Line 1 initializes to `null` the variable *options* that will contain the alternatives for top-level goals. *options* is an array of arrays of plan sets. One plan set is an applicable strategy to achieve a top-level goal; an array of plan sets contains all strategies to achieve a top-level goal; and an array of arrays of plan sets contains all strategies to achieve a set of top-level goals.

The *for* cycle in lines 2-4 iterates over all the current top-level goals of the agent—based on the semantics given in Table 5.1—and populates the variable *options* with the possible variants to achieve the current goals. Line 3 creates an array of plan sets that contains all possible plans for a specific top-level goal. The invoked function GeneratePlans explores the AND/OR goal tree and identifies all valid minimal plans. Minimality means that, whenever an OR-decomposition is encountered, only one sub-goal is selected. In other words, OR-decompositions are treated here as XORs. In line 4 the algorithm adds the possible plans for the examined top-level goal to the variable *options*.

As the cycle terminates, the variable *cartProduct* is initialized to the Cartesian product of the sets in *options*; in such a way, each array of plansets in *cartProduct* is an applicable plan to achieve all top-level goals (line 5). Line 6 initializes the variable *variants* to null; this variable is simply a data structure that allows for simpler handling of alternative than *cartProduct*. Lines 7-11 populate *variants*: for each option (line 7) a `Variant` is declared, the plansets in that option are added to the variant (lines 9-10), and the variant is added to *variants*. Once the array of variants is populated, the algorithm returns it.

We use the example from the smart-home STS shown in Figure 5.2 to illustrate variant generation and selection. The patient has two top-level goals. Both goals have one active instance: he wants to watch a movie and to call his daughter. The former goal is AND-



CONTEXTS:
Patient p, Television tv
c_1: p.sitting=false $\vee$ pp.infrontOfTv=false
c_2: p.onWheelChair=true
c_3: tv.hasOnDemand=true

COMPENSATION COST:

Use mobile phone:	2
Use landline:	2
Sit on wheelchair:	10
Sit on sofa:	6
Select TV movie:	1
Watch DVD:	3

SOFTGOALS PRIORITY:

Comfort:	0.4
Low effort:	0.4
Variety of choice:	0.2

BEST REACTION COST:

Use mobile phone:	3
Use landline:	2
Sit on wheelchair:	6
Sit on sofa:	8
Select TV movie:	1
Watch DVD:	2

Figure 5.2: An example from the smart-home scenario to show variant generation and selection

decomposed to two sub-goals: the patient has to sit in front of the TV and to choose a movie. In case he is already sitting in front of the TV (context c_1 holds), the first sub-goal need not be reached. To sit in front of the TV, two plans are possible: sit on the wheelchair or sit on the sofa. The contributions to soft-goals tell that the first option requires more effort (unless he is already sitting) and is less comfortable. To choose a movie, the patient can either select a TV movie or watch a DVD. These options have different contributions to soft-goal Variety of choice: a TV movie is better if the TV has on-demand features; otherwise, watching a DVD is preferable. To achieve the other top-level goal Call daughter the patient can either use his mobile phone or the land-line. Using a mobile phone is more comfortable, while using the land-line requires less effort.

Let's see how Algorithm 5.3 applies to this scenario. The for cycle in lines 2-4 iterates over the two active top-level goals, and populates the array *options* as follows:

$$
\begin{aligned}
\text{options}[1] = \quad & \{\text{sit on wheelchair, select TV movie}\}, \{\text{sit on wheelchair, watch DVD}\}, \\
& \{\text{sit on sofa, select TV movie}\}, \{\text{sit on sofa, watch DVD}\} \\
\text{options}[2] = \quad & \{\text{use mobile phone}\}, \{\text{use landline}\}
\end{aligned}
$$

Then, the algorithm computes the Cartesian product of the elements of *options*, and populates variable *cartProduct* as follows:

$$
\begin{aligned}
\text{cartProduct}[1] = \quad & \{\text{sit on wheelchair, select TV movie}\}, \{\text{use mobile phone}\} \\
\text{cartProduct}[2] = \quad & \{\text{sit on wheelchair, select TV movie}\}, \{\text{use landline}\} \\
\text{cartProduct}[3] = \quad & \{\text{sit on wheelchair, watch DVD}\}, \{\text{use mobile phone}\}
\end{aligned}
$$

cartProduct[4] = {sit on wheelchair, watch DVD}, {use landline}
cartProduct[5] = {sit on sofa, select TV movie}, {use mobile phone}
cartProduct[6] = {sit on sofa, select TV movie}, {use landline}
cartProduct[7] = {sit on sofa, watch DVD}, {use mobile phone}
cartProduct[8] = {sit on sofa, watch DVD}, {use landline}

Once *cartProduct* is initialized, lines 6-11 copy its content into *variants*, so that the generated variants can be easily processed by the variant selection algorithm.

Algorithm 5.4 Selection of the best reconfiguration variant

```
selectAlternative(Variant [] $alt$, Plan [] $failed$, $started$, $done$)
 1  for each $v$ in $alt$
 2  do $v.cost \leftarrow \sum_{p \in failed} p.compCost$
 3      for each $sp$ in $started$
 4      do if $p$ in GetAllPlans($v$)
 5             then $v.cost$ += $p.compCost$
 6      for each $gt$ in $v.goalTrees$
 7      do for each $pl$ in $gt.plans$
 8          do int $contrib, totPriority \leftarrow 0$
 9             for each $c$ in $pl.contribs$
10             do $contrib$ += $c.val * c.softgoal.priority$
11                $totPriority$ += $c.softgoal.priority$
12             $contrib \leftarrow contrib / totPriority$
13             $v.contrib$ += $contrib /$ Size($gt.plans$)
14             if $pl$ not in $started \cup done$
15               then $v.cost$ += GetBestReaction($pl$)
16  $minCost \leftarrow$ min($v.cost \mid v \in alt$)
17  $maxCost \leftarrow$ max($v.cost \mid v \in alt$)
18  $minCont \leftarrow$ min($v.contrib \mid v \in alt$)
19  $maxCont \leftarrow$ max($v.contrib \mid v \in alt$)
20  $bestVal \leftarrow -\infty$
21  $bestVariant \leftarrow$ null
22  for each $v$ in $alt$
23  do $nCost \leftarrow (v.cost - minCost)/(maxCost - minCost)$
24      $nCont \leftarrow (v.contrib - minCont)/(maxCont - minCont)$
25      if GetAllPlans($v$) $\cap failed \neq \emptyset$
26        then $varValue \leftarrow varValue - 4$
27      $varValue \leftarrow varValue + nCont - nCost$
28      if $varValue > bestVal$
29        then $bestVal \leftarrow varValue$
30             $bestVariant \leftarrow v$
31  return $bestVariant$
```

Algorithm 5.4 selects one variant from a set of variants. Its input consists of an array of variants *alt*—like that generated by Algorithm 5.3—and three arrays of plans with self-explanatory names: *failed*, *started*, and *done*. The *for* cycle in lines 1-15 computes the cost and the contribution to soft-goals for each variant in *alt*.

Lines 2-5 determine the cost of the examined variant, which corresponds to the sum of the compensation cost for failed plans (we assume that each failed plan has to be compensated) and the compensation cost for started plans that will not be in the new configuration. Compensation cost refers to the actions needed to revert or nullify the effects of a failed (or partially executed) plan.

Lines 6-15 iterate the goal trees in the variant and compute contribution to soft-goals. Line 7 iterates all plans in the goal tree, line 8 initializes the variables *contrib* and *totPriority* used in the cycle, whereas line 9 iterates all the soft-goals the examined plan contributes to. In line 10, the contribution value from the plan to the soft-goal is multiplied by the soft-goal priority, and the result is added to variable *contrib*.

In line 11, the soft-goal priority is added to the total priority for the considered plan. Then (line 12) the plan contribution is divided by the total priority to compute the average soft-goal contribution for the examined plan, and (line 13) the contribution value for the whole variant is updated. Lines 14-15 update the reconfiguration cost by adding the minimum reaction cost for the plans that are not started nor done (GetBestReaction) if multiple reactions are available (e.g., both task assignment and system pushing).

Lines 16-19 initialize the variables that contain the minimum/maximum variant cost (16-17) and the minimum/maximum contribution (18-19). Lines 20-21 initialize the variable to contain the best value to $-\infty$ and the best variant to null. The *for* cycle in lines 22-30 performs a min-max normalization of costs and contributions in the $[-1, +1]$ range. This is needed to make the two factors comparable; in such a way, variant selection gives equal weight to both factors. Lines 23 and 24 compute the normalized value for a variant's cost and contribution, respectively.

Line 25 deals with variants that contain a failed plan, and makes any variant without failed plans preferable to all variants including failed plans. The variant value (line 27) is the normalized contribution minus the normalized cost. Lines 28-30 update the variable containing the best variant if the examined variant is better than all variants examined so far. Line 31 returns the best variant.

Let's now illustrate Algorithm 5.5 on the example in Figure 5.2. Suppose the current variant includes plans use mobile phone, sit on wheelchair, and select TV movie; also suppose that sit on wheelchair is failed because the patient fell down while sitting, and that select TV movie is in progress. Let the array of variants *alt* contain the eight variants that were produced by the Cartesian product in Algorithm 5.3. The cycle of lines 2-15 of the variant selection algorithm iterates over all variants and determines, for each variant, (i) the average soft-goal contribution; and (ii) the cost to compensate the effects of the failed and started plans. The results of this phase are as shown in the following table, whose columns represent the variant number (Var), the compensation cost (CCost), the reaction cost (RCost), the overall cost (Cost), and the contribution to soft-goals (Contrib):

Var	CCost	RCost	Cost	Contrib
1	10	9	19	0.167
2	10	8	**18**	0.067
3	10	11	21	**0.433**
4	10	10	20	0.333
5	10	11	21	0.083
6	10	10	20	**-0.027**
7	10	13	**23**	0.350
8	10	12	22	0.250

The bold values in the table highlight the minimum and maximum costs and soft-goal contributions. Now, Algorithm 5.4 enters the cycle of lines 22-30 to identify the best variant. For each variant, it computes a normalized value that considers contribution and cost. In the example, this leads to the following results:

Variant	1	2	3	4	5	6	7	8
Value	-4.007	-3.907	-3.600	-3.617	-0.361	-0.4	**-0.181**	-0.199

The best variant is the seventh. Despite of its cost (which is the highest overall), its contribution to soft-goals makes it preferable to others. This variant is radically different from the current one, for it involves sitting on sofa and watching a DVD, rather than sitting on wheelchair and selecting a TV movie.

5.2.2 A cost-based algorithm for adaptation with commitments

We present an adaptation algorithm that considers not only internal capabilities of the considered agent, i.e. its plans, but also (and mostly) the social relationships with other agents. This algorithm is based on the conceptual model proposed in Section 3.4.2 to deal with social variability. Variants are computed by combining capabilities with commitments the agent can make to and get from other agents.

Like the algorithm presented in Section 5.2.1, this approach consists of variant generation and variant selection. We do not describe variant generation here; such activity consists of finding all variants that satisfy the semantics presented in Definition 3.16. We provide a concrete way to generate variants in Chapter 6 (Algorithm 6.1). There, our prototype encodes the variant generation problem to a satisfiability problem and feeds an external reasoner based on SAT-solving techniques.

We focus here on the post-processing that is carried out after variant generation. Such post-processing consists of filtering the generated variants and selecting the best variant.

Filtering is needed to deal with negative contributions, since the external reasoner used for variant generation deals only with Horn clauses, and negative contributions cannot be mapped to Horn clauses. Additionally, the algorithm performs further processing to consider the adaptation policy defined by system administrators. Our algorithm supports policies including the following factors:

– *Variant selection criteria* refers to the algorithm to determine the best variant. We currently support two algorithms. The first one (`overallCost`) considers the overall variant cost, and selects the variant having minimal total cost. The second one (`delta`) takes into account the current variant, and computes the cost as the delta between the new possible variant and the current one. The first criteria is similar to planning from scratch, whereas the second one is a replanning algorithm.

– *Variant exclusion factors* specify typical criteria an agent could consider to rule out some variants from the generated ones. First, an agent might want a different solution (`noRetry`). Selecting such option corresponds to preventing a retry strategy (i.e. it forces an adaptation, as in Definition 3.3). Second, the agent might want to avoid threatened capabilities and commitments (`avoidThreatened`). Also, failed capabilities and violated commitments might be excluded (`avoidFailed`).

– *Compensation cost* represents how expensive is for an agent to revert/nullify the effects of a capability that is currently exploited and that will not be in the next variant. Compensation cost is considered when determining variant cost, if such option is selected (`compensationCost`).

– *Third-party commitments* are those that do not involve the agent neither as debtor nor as creditor (`useThirdPartyCommitments`). The agent can exploit these commitments to generate variants. The rationale is that, if the agent brings about the antecedent and communicates that to the debtor, the debtor would be unconditionally committed for the consequent to the creditor. Notice that such option is considered during variant generation, not during selection. It will appear in Algorithm 6.2 (Chapter 6).

– *Opportunity threshold* defines when an opportunity should be adopted. The rationale behind this option is that, if the current variant is not at risk, the agent needs a clear incentive to switch to another variant. If the variant selection strategy is chosen, this threshold is how less the new variant would cost, in percentage. If the delta strategy is chosen, opportunities are taken only if their delta is lower than a fixed value.

Algorithm 5.5 Cost-based variant selection

```
FilterVariants(Variant [] $variants$, Agent $ag$, Policy $pol$, Event [] $triggers$, Variant $currVar$)
 1  $Variant$ [] $toReturn \leftarrow$ null
 2  for each Variant $v \in variants$
 3  do if ($pol.noRetry$ and $v = currVar$) then $goToNextVariant$
 4     SoW [] $supported \leftarrow$ ComputeSupported($ag, v$)
 5     for each Goal $g \in v.capabilities$
 6     do if ($pol.avoidThreatened$ and $g.status =$ threatened) or ($pol.avoidFailed$ and $g.status =$ failed)
 7           then $goToNextVariant$
 8        if $pol.simType =$ overallcost
 9           then $v.solCost \leftarrow v.solCost + g.cost$
10           else if $g.status =$ idle or $g.status =$ failed
11                   then $v.solCost \leftarrow v.solCost + g.cost$
12     for each Commitment $c \in v.commitments$
13     do if ($pol.avoidThreatened$ and $c.status =$ threatened) or ($pol.avoidFailed$ and $c.status =$ violated)
14           then $goToNextVariant$
15        if $pol.simType =$ overallcost
16           then $v.solCost \leftarrow v.solCost + c.cost$
17           else if $c.status =$ idle
18                   then $v.solCost \leftarrow v.solCost + c.cost$
19     if $\nexists$Event $e \in triggers : e.type =$ threat
20       then if not OpportunityExploitable($v, pol$) then continue
21     SoW [] $scoped \leftarrow$ null
22     for each SoW $s \in supported$
23     do if $s \notin scoped$ then PropagateSupport($s, scoped$)
24     for each Contribution $c \in$ GetNegContributions($ag$)
25     do if $c.from \in supported$ and $c.to \in supported$
26           then PropagateNegative($c.to, supported$)
27     if not SupportsQuery($supported, ag.currentGoals$) then continue
28     if $pol.considerCompensation$
29       then for each SoW $s \in currVar.supported$
30              do if $s \notin supported$ then $v.solCost \leftarrow v.solCost + s.compensationCost$
31              for each Commitment $c \in currVar.commitments$
32              do if $c \notin v.commitments$ then $v.solCost \leftarrow v.solCost + c.compensationCost$
33              if $\nexists$Event $e \in triggers : e.type =$ threat
34                then if not OpportunityExploitable($v, pol$) then continue
35     AddElement($toReturn, v$)
36  return $toReturn$
```

Function FilterVariants (Algorithm 5.5) filters the generated variants to exclude those that do not match the adaptation policy. Its input consists of the set of variants to be filtered (*variants*), the agent *ag*, the adaptation policy *pol*, the events that have triggered the adaptation process (*trigger*), and the current variant *currVar*.

Line 1 defines the array to contain the filtered variants (*toReturn*). The cycle of lines 2-34 iterates over the input variants and filters them, selecting only those matching the adaptation policy. Line 3 enacts the `noRetry` policy: if the generated variant is like the current one, the variant is ignored and the cycle continues with the next variant. Line 4 calls algorithm ComputeSupported to derive the states of the world that the considered variant supports, i.e. those that will eventually hold if the variant is enacted correctly and no unexpected changes in the world occur. This algorithm is formalized in Algorithm 5.6 and will be described later in this section.

The *for* cycle of lines 5-11 begins variant cost computation by considering the involved capabilities. The capabilities are iterated. For each capability, different options are possible. If the adaptation policy excludes threatened (failed) capabilities, and the examined capability is threatened (failed), the current variant is ruled out and the variants iteration cycle continues with the next variant (lines 6-7). If this is not the case, cost computation is performed. If the policy is based on the overall variant cost, the cost of the capability is added to the variant cost (lines 8-9). If the delta policy is chosen, the cost is added only if the capability is currently not exploited or failed (lines 10-11).

The cycle of lines 12-18 adds the cost of the commitments in the variant. Lines 13-14 rule out the variant if the commitment is violated (threatened) and the adaptation policy specifies to exclude variants involving violated (threatened) commitments. Lines 15-18 update the variant cost: if the overall cost policy is chosen, the cost is always added; if the delta policy is chosen, the cost is added only if the commitment is not active. Note how violated commitments do not imply additional costs, for no plan has to be carried out by the considered agent. Relying on violated commitments is a sensible option if a delay is expected, while it is very unlikely to succeed in case of cancellation.

Lines 19-20 consider variant filtering in presence of opportunities only, i.e. when adaptation is not triggered by any threat. In such case, the variant is not selected if the opportunity does not provide a significant advantage over the current variant. This is verified by invoking function OpportunityExploitable (Algorithm 5.7), that we will describe later in this section.

Lines 21-23 populate the array of states of the world *scoped*. For each state of the world supported by the current variant, the function PropagateSupport is invoked to compute the scoped states of the world, according to the semantics of scoped we provided in Definition 3.15. Then, lines 24-26 take into account negative contributions between scoped states of the world by invoking PropagateNegative, which will remove from the set of supported goals those having an incoming negative contribution from a supported goal. Both propagation algorithms are described later in this section.

If the updated set of supported states of the world does not satisfy the goal query—the goals the agent wants to achieve—the variant is not selected (line 27). If the adaptation policy includes compensation cost (line 28), the variant cost is updated with the compensation cost for the goals and commitments that are in the current variant and are not in the next possible variant (lines 29-32). Lines 33-34 perform the same check done in lines 18-19 for adaptations triggered by opportunities. Repeating the analysis is necessary because the compensation cost has been added, thus the variant might have become not exploitable now.

At the end of the cycle the current variant is added to the set of filtered variants (line 35), since no rule in the policy prevented it from being selected. Line 36 concludes the algorithm by returning the filtered variants.

Algorithm 5.6 Computation of supported goals in a variant

```
ComputeSupported(Agent $ag$, Variant $var$)
 1  SoW [] $supported \leftarrow$ null
 2  for each Goal $g \in var.capabilities$
 3  do AddElement($supported$, GetState($g$))
 4  for each SoW $s \in var.states$
 5  do AddElement($supported$, $s$)
 6  for each Commitment $c \in var.commitments$
 7  do if $c.debtor = ag$
 8        then AddElement($supported$, GetSymbols($c.antecedent$))
 9        else AddElement($supported$, GetSymbols($c.consequent$))
10  return $supported$
```

Algorithm 5.6 describes function ComputeSupported. Its input consists of an agent and a variant. Line 1 initializes the array to contain supported states of the world. Lines 2-3 iterate over the capabilities in the variant, and adds to the set of supported states those corresponding to the capabilities. Lines 4-5 adds the states of the world that already hold to the set of supported states. Lines 6-9 iterate over the commitments in the variant. If the agent plays debtor in the commitment, all symbols in the antecedent are added to the set of supported states. If the agent plays creditor, all symbols in the consequent are added. Finally (line 10), the algorithm returns the supported states.

Algorithm 5.7 Checking if an opportunity should be exploited

```
OpportunityExploitable(Variant $var$, Policy $pol$)
1  if $pol.threshold = 0$ then return true
2  if $pol.simType =$ overallcost
3    then if $var.solcost <$ (currentsolcost $- pol.threshold\%$)
4           then return true
5    else if $var.solcost < pol.delta$
6           then return true
7  return false
```

Algorithm 5.7 (OpportunityExploitable) is a boolean function that determines if a certain variant is exploitable as an opportunity, by checking if its cost matches the adaptation policy. If no threshold is specified, then the variant is always exploitable (line 1). Otherwise, two cases are possible: (a) if the policy considers the overall cost, the variant can be exploited if its cost is minor than the current solution variant minus the specified threshold percentage (lines 2-4); (b) if delta cost is selected, the variant is exploitable if its cost is minor than the specified delta threshold (lines 5-6). If none of these conditions holds, the variant cannot be exploited (line 7).

Algorithm 5.8 Support propagation in a goal model

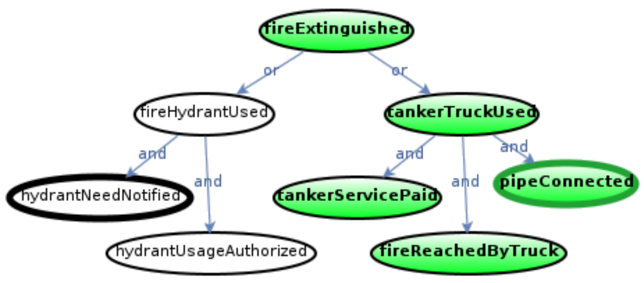
```
PropagateSupport(SoW state, SoW [] scoped)
1 AddTo(scoped, state)
2 if IsGoal(state) and GetParent(state) $\neq$ null and GetParent(state) $\notin$ scoped
3   then Goal parent $\leftarrow$ GetParent(state)
4        if parent.andDecomposed
5          then if $\forall$Goal $g \in parent.children : g \in scoped$
6                 then PropagateSupport(parent, scoped)
7          else PropagateSupport(parent, scoped)
8 for each SoW state1 $\in$ GetPosContributionsFrom(state)
9 do if state1 $\notin$ scoped then PropagateSupport(state1, scoped)
```

Algorithm 5.8 (PropagateSupport) propagates support. It is a recursive algorithm that updates its input parameter *scoped* starting from the input state of the world. First, the state itself is added to the set of supported states (line 1). Then, if the state corresponds to a goal, it is not a root level goal, and the parent does not already belong to *scoped*, support is propagated bottom-up in the goal tree (lines 2-7). If the parent is AND-decomposed (line 4), support is propagated only if all its children goals are supported (lines 5-6). If the parent is OR-decomposed, support is propagated since the considered goal (its child) is supported (line 7). Lines 8-9 perform support propagation for all goals that are positively contributed by the considered goal.

Algorithm 5.9 Negative contribution propagation in a goal model

```
PropagateNegative(SoW from, SoW [] supported)
1 if from $\in$ supported
2   then RemoveElement(supported, from)
3        if IsGoal(state) and GetParent(state) $\neq$ null and GetParent(state) $\in$ supported
4          then Goal parent $\leftarrow$ GetParent(state)
5               if parent.orDecomposed
6                 then if $\nexists$Goal $g \in parent.children : g \in supported$
7                        then PropagateNegative(parent, supported)
8                 else PropagateNegative(parent, supported)
```

Algorithm 5.9 (PropagateNegative) propagates the effect of negative contributions in a goal model. Its input parameters are a state of the world *from* and the set of supported states of the world. The algorithm is initially invoked for goals that have negative contributions to other states. If the considered state does not belong to such set, the algorithm terminates. Otherwise, the considered state is removed (line 2) and further processing is performed (lines 3-8). If the state corresponds to a goal, it is not a top-level goal, and its parent belongs to supported (line 3), two options are possible. If the parent is OR-decomposed and no sub-goal is supported, the algorithm is recursively invoked on the parent (lines 5-7). If the parent is AND-decomposed, the algorithm is always recursively invoked.



Current commitments: C_2, C_3

(a)

Element	Acost	Ccost
hydrantNeedNotified	6	4
pipeConnected	17	8
C_1	15	7
C_2	8	0
C_3	13	20
C_4	31	16

(b)

$$C_1 = C(\mathsf{Brigade1}, \mathsf{Jim}, \mathsf{hydrantNeedNotified}, \mathsf{hydrantUsageAuthorized})$$
$$C_2 = C(\mathsf{Jim}, \mathsf{Brigade1}, \mathsf{tankerServicePaid}, \mathsf{tankerTruckUsed})$$
$$C_3 = C(\mathsf{Tanker1}, \mathsf{Jim}, \mathsf{tankerServicePaid}, \mathsf{fireReachedByTruck})$$
$$C_4 = C(\mathsf{Tanker2}, \mathsf{Jim}, \mathsf{tankerServicePaid}, \mathsf{fireReachedByTruck})$$

(c)

Table 5.2: A fire-fighting scenario to illustrate variant selection: (a) The current variant for Jim; (b) capabilities and commitments cost; (c) commitments in the scenario

We illustrate now Algorithms 5.5-5.9 on an example from the emergency response scenario (Table 5.2). The goal model and current variant of agent Jim are shown in part (a). He currently intends to fight fire using a tanker truck, which involves getting tanker service paid by Brigade1 via C_2, having fire reached by a tanker truck via C_3, and using the capability for connecting the water pipe to the tanker truck. Part (b) shows the activation cost (ACost) and compensation cost (CCost) for capabilities and commitments. Part (c) describes the commitments made by Jim and the other agents in the scenario (Brigade1, Tanker1, Tanker2).

Suppose commitment C_3 made by the tanker truck service Tanker1 is threatened. This happens because Tanker1 notifies it will be late in reaching the fire due to a traffic jam. C_3 is part of Jim's current variant for goal fireExtinguished. Consequently, such variant is threatened. In response to such threat, an adaptation process is triggered. Jim is using an adaptation policy including variant selection based on the delta criteria, and prescribing to avoid threatened/violated commitments. Jim can currently choose between three variants to extinguish fire:

– exploit his capability for hydrantNeedNotified and get commitment C_1 from Brigade1 to support goal hydrantUsageAuthorized. Bringing about the antecedent of C_1 will make Brigade1 unconditionally committed to authorize hydrant usage.

– make commitment C_2 to Brigade1 to support tankerServicePaid, chain such commitment to C_3 so that Tanker1 is unconditionally committed to fireReachedByTruck, use his capability for pipeConnected. Notice that this is the current variant, which is already partially enacted.

– the same strategy as the previous, but relies on commitment C_4 instead of C_3. This corresponds to making Tanker2 unconditionally committed to reach the fire.

Let's take a look at how Algorithm 5.5 rules out variants that do not match the adaptation policy and how it computes variant cost. We presume that, after variant filtering, agent Jim selects the variant having the lowest cost.

The first variant involves commitment C_1 and Jim's capability for hydrant need notification. In line 3, function COMPUTESUPPORTED computes the states of the world that the capabilities and commitments in the variant support. The returned set consists of hydrantNeedNotified, hydrantUsageAuthorized, fireHydrantUsed, and fireExtinguished. Then, lines 4-10 determine the cost of the capabilities in the variant. Here, there is only one capability (hydrantNeedNotified), whose activation cost is 6. The following lines 11-17 add the cost of involved commitments, which corresponds to the activation cost of C_1, 15. Lines 20-26 are not relevant here, for there are no contributions in the model. Then, lines 27-33 determine the compensation cost. This includes the cost of capability pipeConnected (8), commitment C_2 (0), and commitment C_3 (20). The variant is not filtered out, and its overall cost is 49.

The second variant (C_3, C_2, pipeConnected) supports (line 3) states tankerServicePaid, fireReachedByTruck, pipeConnected, tankerTruckUsed, and fireExtinguished. Cost computation for capabilities (lines 4-10) adds no cost, for pipeConnected is already in use (the *if* control in line 9 excludes it). Then, cost calculation for commitments is started (lines 11-17), and the variant is filtered out due to the adaptation policy. Indeed, commitment C_3 in the variant is threatened, and the *if* control in line 12 makes the algorithm continue with the next variant.

The third variant (C_4, C_2, pipeConnected) supports the same states as the second one. Cost computation for capabilities is also the same and adds no cost. Cost computation for commitments (lines 11-17) adds the activation cost of C_4, 31. The next relevant part of the algorithm is cost computation for compensations (lines 27-33), which adds the compensation cost of C_3 (20). Indeed, C_3 is in the current variant but not in the analysed variant. The variant is added to the set of filtered variants (line 35); its overall cost is 51.

Jim will therefore choose—according to his current adaptation strategy—the first variant. To enact it, he will have to perform several steps. First, he has to revert the effects of the current variant: cancel commitment C_2 he made to Brigade1, release Tanker1 from commitment C_3, compensate pipeConnected. Second, he has to carry out some actions to adopt the new variant: use his capability for hydrantNeedNotified to detach C_1 and make Brigade1 unconditionally committed to hydrantUsageAuthorized.

5.3 Adaptation patterns for socio-technical systems

In the previous two sections we have detailed diagnosis and reconfiguration algorithms. Here, we focus on adaptation patterns for socio-technical systems. This type of patterns captures common tactics to cope with threats and opportunities that arise in STSs. Our patterns have a clear focus on the interactions the considered agent should engage in to achieve his goals.

The concept of pattern has a long-standing tradition in software engineering, and it is widely recognized as a fundamental design abstraction to reuse successful solutions to common problems [GHJV95]. Our purpose is similar to that of design patterns in software design: we want to identify and represent adaptation strategies that an agent would typically perform in response to threats and to exploit opportunities. We use the term “tactics” instead of “patterns”, for they represent an agent’s behaviour to better deal with the situation at-hand.

We describe most tactics both textually and graphically. The latter representation is based on the emergency response setting, and refers to the example in Table 5.2. Each of the Figures 5.3–5.6 depicts Jim’s active variant for his goal fire extinguished before (on the left of the dark, solid arrow) and after adaptation (on the right of the arrow).

A variant depicts the active part of Jim’s goal model: those goals in the goal model of Figure 3.16 that Jim has instantiated (we are abusing the Tropos notation by using it to denote the active goals), and the capabilities and commitments required to *support* that goal. Commitments are represented by labelled directed arrows between agents—the debtor and creditor are indicated by the tail and the head of the arrow, respectively.

Our adaptation tactics represent adaptation at a high-level of abstraction: they do not tell which are the specific plans and actions that should be performed to switch from the current variant to a new one. They focus on the essence of adaptation, and show that different goals are supported, new capabilities are exploited, commitments are taken from and made to other agents.

Tactic 1 (Alternative goals) (Example 11) Choose a different set of goals in a goal model to satisfy target goals. The agent believes the current strategy is not likely to succeed, and reacts by modifying its internal strategy.

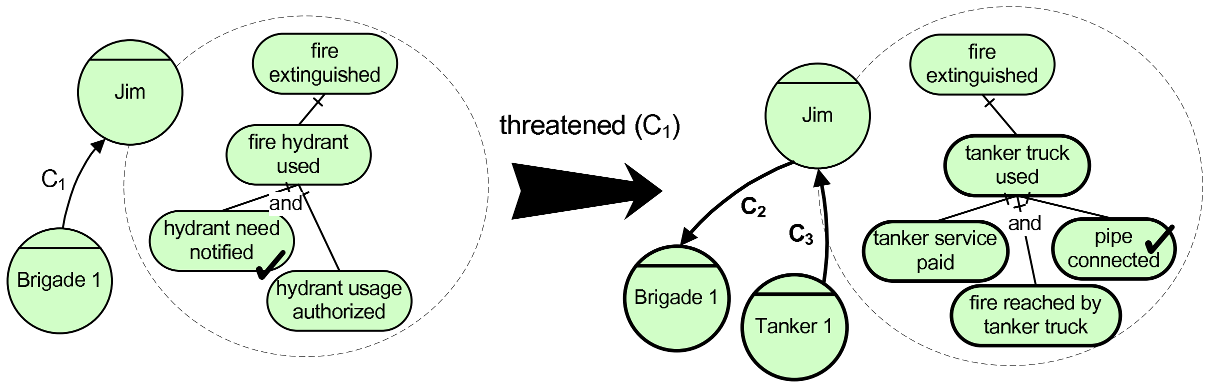


Figure 5.3: Alternative goals: Jim switches from a variant involving fire hydrant usage to another involving tanker truck usage

Example 11 (Figure 5.3) Jim tries to achieve fire extinguished via a variant that relies upon using the fire hydrant. However, the fulfilment of C_1, which is necessary to support the goal, is threatened because Brigade 1 hasn't authorized hydrant usage yet. In other words, Jim's internal policy states that this specific situation corresponds to a threat for C_1. Jim switches to another variant that supports fire extinguished via the alternative goal tanker truck used. This corresponds to selecting an alternative set of goals; also, it requires to make and get commitments to support tanker truck used: Jim makes C_2 to Brigade 1 and gets C_3 from Tanker 1.

Tactic 2 (Goal redundancy) (Example 12) Select a variant that includes redundant ways to satisfy goals. Useful for critical goals that the agent wants to achieve at any cost.

Example 12 (Figure 5.4). Jim's current strategy is to fight the fire via the hydrant. However, C_1 is threatened. So Jim adopts a strategy which involves *also* calling a water tanker truck. By contrast, Example 11 involves no redundancy. Tactics based on goal redundancy are more expensive than others. However, many realistic strategies rely on redundancy For instance, redundancy is a fundamental feature in avionics.

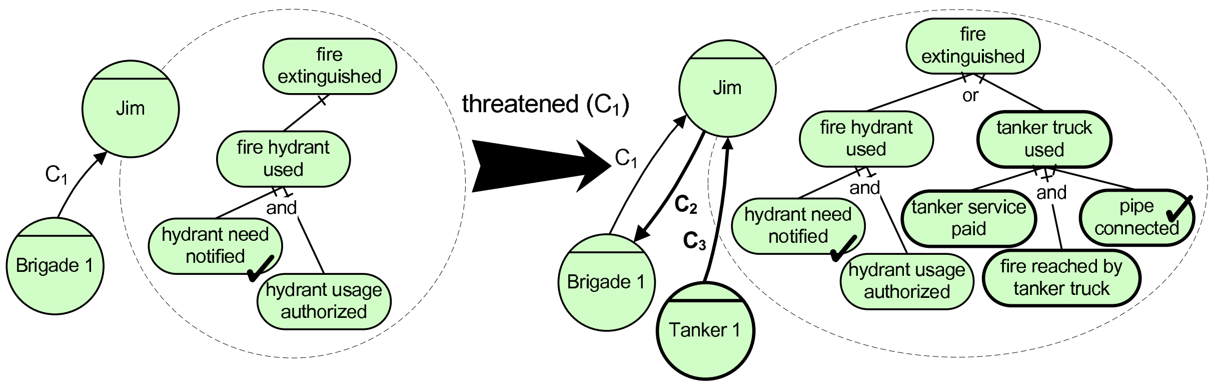


Figure 5.4: Goal redundancy: Jim adopts a redundant variant, which involves also calling a water tanker truck

Tactic 3 (Commitment redundancy) (Example 13) More commitments for a goal are taken. Useful if the agent does not trust some agent it interacts with. Also, it applies when a commitment from another agent is at risk due to the surrounding environment, and a different commitment is more likely to succeed.

Example 13 (Figure 5.5) Jim doesn't trust Tanker 1 much for C_3. He might be interacting with that tanker because it was the only available one. Now another tanker is available, which he trusts more. Jim decides to get a similar commitment C_4 from Tanker 2.

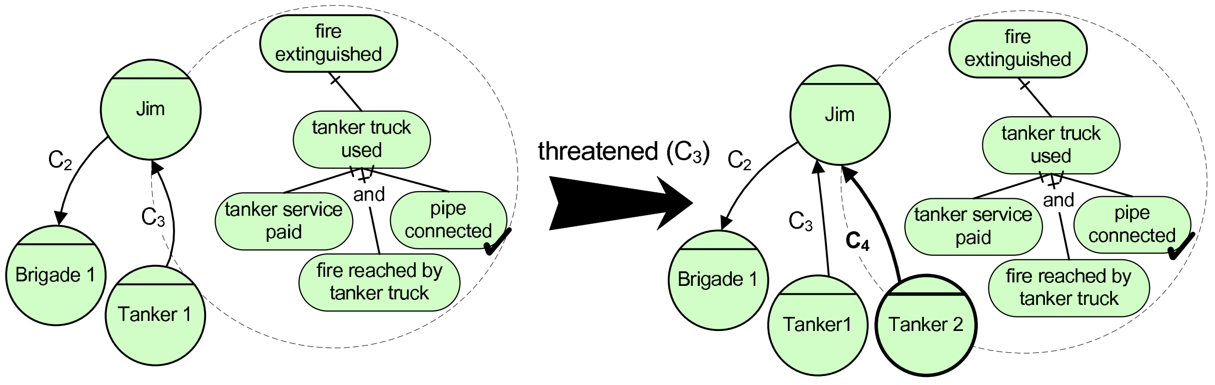


Figure 5.5: Commitment redundancy: Jim gets C_4 from Tanker 2

Tactic 4 (Switch debtor) (Example 14) Get a commitment for the same state of the world but from a different debtor agent. Useful if the creditor believes the current debtor will not respect its commitment or a more trustworthy debtor comes into play. Unlike commitment redundancy, the original debtor is released from his commitment.

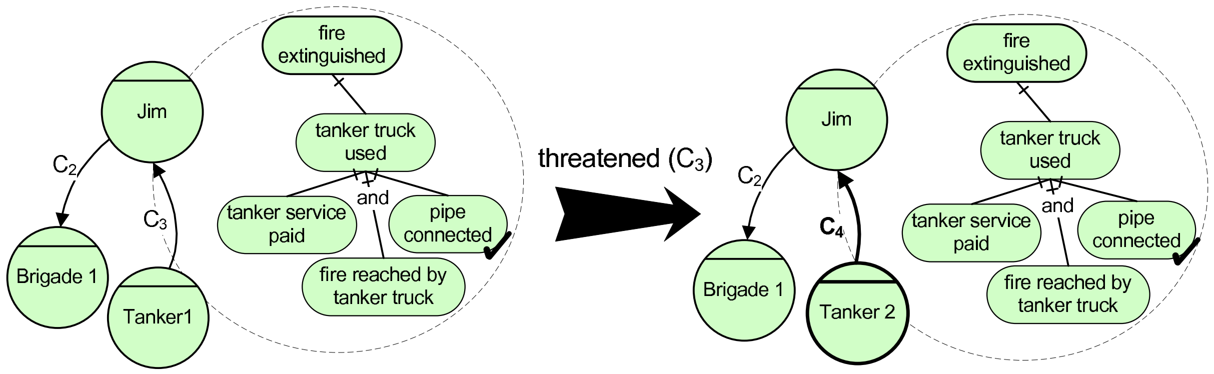
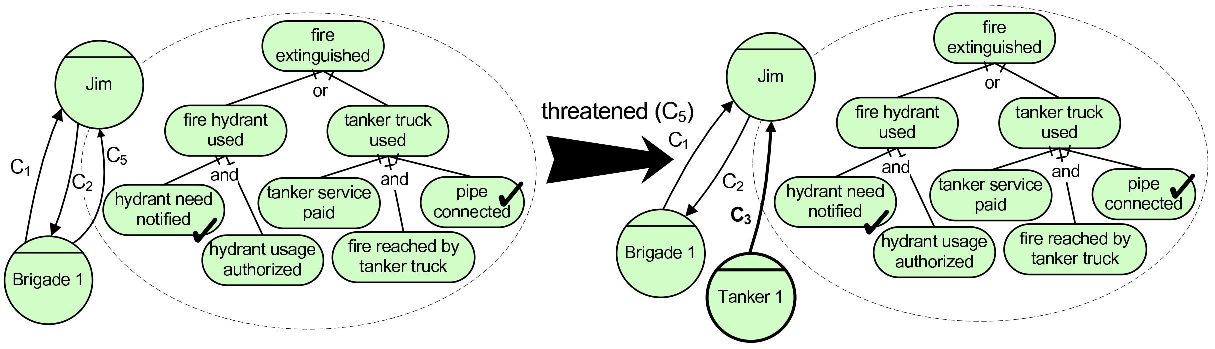


Figure 5.6: Switch debtor: Jim releases Tanker 1 from C_3 and takes C_4 from Tanker 2

Example 14 (Figure 5.6). Jim takes C_3 from Tanker 1, but fears that the Tanker 1 will violate the commitment. Therefore Jim releases Tanker 1 from C_3 and instead gets C_4 from Tanker 2.

Tactic 5 (Division of labor) (Example 15) Rely on different agents for different goals instead of relying on a single agent. Distribution of work reduces the risk of complete failure by splitting the task.



C_5=C(Brigade 1, Jim, tanker service paid, fire reached by tanker truck)

Figure 5.7: Division of labour: Jim releases Brigade 1 from C_5 and takes commitment C_3 from Tanker 1

Example 15 (Figure 5.7) Suppose Jim wants to use both fire hydrant and a water tanker truck. Also, suppose Brigade 1 acts as a water tanker provider (see Commitment C_5 in the figure). Jim is currently relying on the tanker service provided by Brigade 1. However, Jim applies division of labour to minimize risk of failure: he releases Brigade 1 from C_5 and takes commitment C_3 from Tanker 1.

Tactic 6 (Commitment delegation) (Example 16) An agent delegates a commitment in which he is debtor to another agent, perhaps because he can't fulfil it and does not want to violate his commitment.

Example 16 Jim does not have resources to fight a fire, for they are already assigned to other tasks, e.g. a flooding in the area. So he delegates his commitment to extinguish a fire to another fire chief Ron of a neighbouring town.

Tactic 7 (Commitment chaining) (Example 17) Agent x's commitment $\mathsf{C}(x, y, g_0, g_1)$ is supported if he can get $\mathsf{C}(z, x, g_2, g_1)$ from some z and if x supports g_2. This tactic is very common: x commits to another agent y for some g_1 so that, as the antecedent g_0 is brought about by y, agent x can take another agent from z and support his goal g_2.

Example 17 Jim wants to achieve goal tanker truck used. It makes C_2 to Brigade 1 so that tanker service paid is achieved. In such a way, as soon as commitment C_2 is detached, he can get C_3 from Tanker 1 and achieve his goal to use a tanker truck.

5.3.1 Variant selection and operationalization

The adaptation tactics provided in Section 5.3 are a valuable resource for the development of adaptive agents. However, they do not detail how they are applied. To overcome such limitation, we discuss here different criteria to select a variant (Section 5.3.1) and enact/operationalize it (Section 5.3.1). Some of the variant selection criteria are exploited by the algorithms in Section 5.2.

From Definition 3.16, a variant V is an abstract strategy. It is a triple $\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor$ composed of goals $\mathcal{G}$, commitments $\mathcal{P}$, and capabilities $\mathcal{C}$. Neither commitments nor capabilities are grounded to concrete entities. The agent should therefore operationalize the variant: commitments must be bound to actual agents, capabilities to real plans.

Variant selection

Variant selection is the choice of one variant among the possible ones. Function selVariant takes as input a set of variants and a state of the world and returns one of these variants. Taking into account the state of the world is important because the agent need not identify ways to achieve a sub-goal if the corresponding state of the world already holds. Also, the state of the world might make some solutions non-adoptable.

$$\textsc{selVariant} : 2^V \times S \to V$$
$$\textsc{selVariant}(\{V_1, \ldots, V_n\}, \sigma) = V_i : 1 \leq i \leq n$$

Table 5.3 shows some common criteria for variant selection. Due to its autonomy, each agent is free to choose its own criterion. Criteria can be merged to define more specific selection procedures. For example, an agent might want to consider both cost and stability (this is the case of Algorithm 5.4).

Name	**Description**
Cost	Minimize the overall cost, expressed as money, needed resources, time
Stability	Minimize the distance between the current strategy and the new one
Soft-goals	Maximize the satisfaction of quality goals (performance, security, risk)
Preference	Choose preferred goals and commitments
Goal Redundancy	Choose a redundant variant to achieve critical goals

Table 5.3: Generic criteria for variant selection

Event	**Condition**	**Action**
threatened(C_1)	target(fire extinguished), $\neg$made(C_2), $\neg$taken(C_3), $\neg$taken(C_4), adopted(fire hydrant used), $\neg$adopted(tanker truck used)	adopt(tanker truck used), adopt(tanker service paid), adopt(fire reached by tanker truck), adopt(pipe connected), get(C_z), useCapability(pipe connected), make(C_y)

Table 5.4: Event-Condition-Action rule for variant selection with goal redundancy (Figure 5.4)

Example 18 Table 5.4 specifies the function selVariant for Figure 5.4 as an Event-Condition-Action rule. Such a function is based upon goal redundancy (Tactic 2). The triggering event is that commitment C_1 is threatened. It applies if the target goal is fire extinguished, commitments C_2, C_3, C_4 are not in place, Jim adopted goal fire hydrant used and not tanker truck used. The action specifies the transition to the new variant. Jim adopts goal tanker truck used and its children, uses his capability for pipe connected, gets commitment C_z, and commits for C_y. C_y and C_z are the unbound commitments introduced in Example 4: $\mathsf{C}_y = \mathsf{C}(\mathsf{Jim}, y, \mathsf{tanker\ service\ paid}, \mathsf{fire\ extinguished})$, $\mathsf{C}_z = \mathsf{C}(z, \mathsf{Jim}, \mathsf{tanker\ service\ paid}, \mathsf{fire\ reached\ by\ tanker\ truck})$.

Variant operationalization

The operationalize function takes as input the selected variant and an agent's state and returns a set of states.

$$\textsc{operationalize} : V \times S \rightarrow 2^S$$

$$\textsc{operationalize}(\lfloor \mathcal{G}, \mathcal{P}, \mathcal{C} \rfloor, \sigma) = \textsc{bindToPlan}(\mathcal{C}); \textsc{bindAgent}(\mathcal{P})$$

Operationalization means identifying a concrete strategy to achieve goals and commitments in a variant. Commitments are bound to real agents (BINDAGENT), whereas capabilities are bound to executable plans (BINDPLAN). Let's explain why such a function returns a set of states instead of a single one. Suppose Jim's selected variant includes finding some agent that will commit for C_z. Jim may send a request message for C_z to all known tanker providers—Tanker 1 and Tanker 2—and get a commitment from the first one that accepts. If Tanker 1 answers first, the function returns a state σ_1 where C_3 holds; if Tanker 2 answers first, the returned state will be σ_2 such that C_4 holds.

With respect to Table 5.4, operationalization would be invoked inside useCapability(pipe connected) to bind an appropriate plan to the capability, and inside get(C_z) and make(C_y) to bind z and y to the appropriate agents.

Name	**Description**
Comm Redundancy	More commitments for the same goal from different agents
Division of Labour	Involve many agents, each agent commits for a small amount of work
Delegation	Delegate some commitment where the agent is debtor to someone else
Trust	The agent gets commitments only from other agents it trusts
Reputation	Rely on reputation in community to select agents to interact with

Table 5.5: Generic criteria for variant operationalization

Table 5.5 shows some generic criteria an agent can exploit and combine to operationalize a variant. Operationalization policies can mix these criteria to define fine-grained strategies.

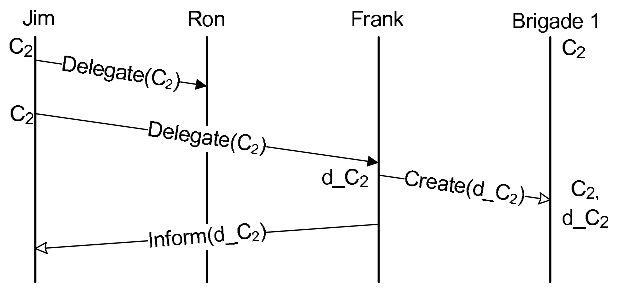


Figure 5.8: Bind commitments to agents: Jim delegates C_2 on the basis of trust

Example 19 Let's operationalize the variant in Example 18. Jim wants to delegate firefighting with tanker truck to the agent he trusts more. He knows two fire chiefs, Ron and Frank. The one he trusts more is Ron.

Step 1. Bind capabilities to plans. Jim binds his capability for pipe connected to a specific

plan where he connects a water pipe to the rear connector of a water tanker truck.
Step 2. Bind commitments to agents. Jim delegates C_2 to Ron, but he doesn't get any response. Thus, he delegates such commitment to Frank, who accepts delegation. Frank creates a commitment to Brigade 1 and notifies Jim. Figure 5.8 illustrates binding to agents.

5.4 Chapter summary

In this chapter we have proposed algorithms for the adaptation control loop of our architecture. In Section 5.1 we have described diagnosis algorithms to identify failures and under-performance. In Section 5.2 we have proposed algorithms to generate possible variants and select the best one. Specifically, we have devised: (i) an algorithm for settings with contextual variability based on soft-goals; and (ii) a cost-based algorithm for settings with social variability. In Section 5.3 we illustrated some adaptation tactics that illustrate the specific features of adaptation in STSs.

The main contribution of this chapter is in the proposed algorithms. They cover the fundamental phases of adaptation and are based on the modelling frameworks of Chapter 3. The diagnosis algorithms enable to detect goal failure, plan failure, and commitment violation. The two reconfiguration algorithms are based on general criteria and apply to different types of STS. The first algorithm deals with contextual variability. It performs variant selection on the basis of two factors: (i) contribution to soft-goals and (ii) configuration stability, i.e. minimize changes between the current variant and the new one. The second algorithm is for STSs with social variability, where available agents and the services the offer—the *commitments* they make—change over time. It supports two variant selection strategies: (i) minimization of the overall solution cost, and (ii) maximization of stability by choosing the less expensive set of changes.

In Section 5.3 we have analysed adaptation in STSs from a broader perspective. We have shown how adaptation in STSs is affected by social factors, and how adaptation tactics rely on engaging in different social relations. On top of these tactics, we proposed a general framework for adaptivity that distinguishes between the selection of an abstract strategy and its operationalization.

Chapter 6

Prototype implementations

We describe two prototype implementations of the conceptual architecture we presented in Section 4.2. They demonstrate the feasibility of our approach. Each implementation focuses on specific aspects of software self-adaptation in STSs. The first implementation is for STSs with a volatile physical context, wherein changes are a threat for the agent's current variant. The second implementation deals with the effect of social variability on variant generation and selection.

6.1 Prototype for settings with contextual variability

The first prototype we developed is meant for socio-technical systems characterized by contextual variability. Examples of these settings are smart-homes and emergency response coordination. For example, in a smart-home, the health conditions of a patient vary over time, and the applicability of different variants depends on the patient's health. Our prototype deals mainly with failures and context changes that make the current variant inadequate to achieve current goals.

We used Java 1.6 as main programming language, the Eclipse Modeling Framework (EMF)[1] to define the meta-model for the requirements models, the DLV-complex reasoner [CCIL08] to support diagnosis and reconfiguration, and the H2 embedded database[2] to keep track of the effect of context changes on active goals and plans. The main features of this prototype are the following:

- The meta-model for requirements models is an independent artefact. Being defined using EMF, it is decoupled from code. Consequently, it is reusable in other applications and is an useful resource for documentation. The resulting implemented architecture is therefore fully model-driven.

[1]http://www.eclipse.org/modeling/emf/

[2]http://www.h2database.com/

– The MDRC cycle is fully implemented. A monitoring component listens for events from connected event sources (either sensors or agents); diagnosis is performed with the aid of the automated reasoning tool DLV-complex; the same tool is used as a planner to generate possible variants; compensation is actuated by controlling effectors in the environment and interacting with other agents.

– The architecture can deal with failures and context changes. The supported failure types are (i) plan failure, i.e. an action is carried out though its critical precondition does not hold; (ii) plan timeout, i.e. an action in the plan does not occur within its deadline; (iii) goal timeout, i.e. a goal is not achieved within its deadline; (iv) inapplicable plan, i.e. the current variant is not applicable any more due to a change in the environment; (v) domain assumption violation, i.e. the system itself or some other agent in the STS perform an action that violates a domain assumption.

– A simulation editor enables to test the architecture's capabilities to respond to failures and exceptional scenarios. Designers can use the simulation editor to define traces that—during simulated sessions—generate events sent to the monitor.

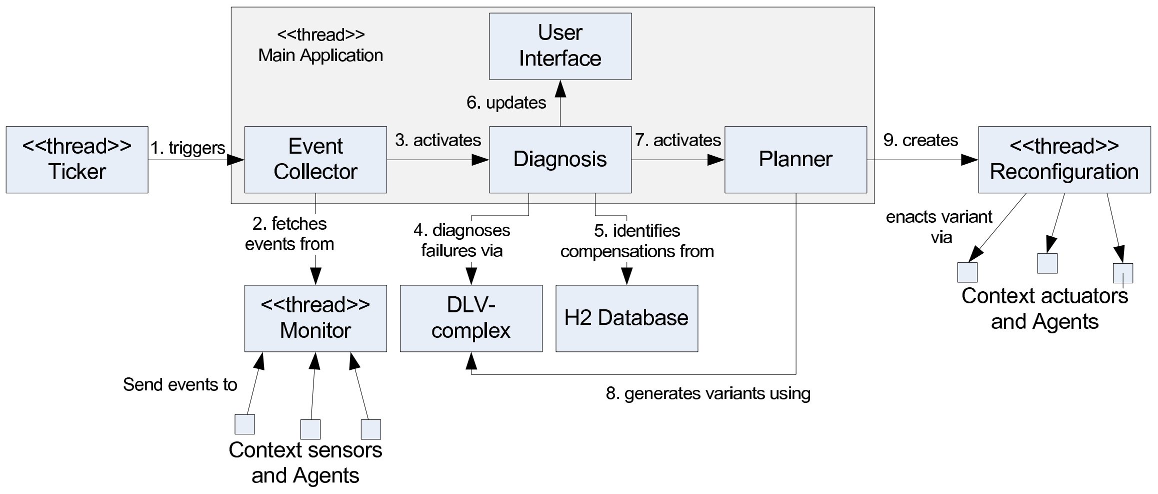


Figure 6.1: Runtime operation of the prototype for STSs with contextual variability

Figure 6.1 shows the basic runtime operation of our implementation. The main application threat consists of an `User Interface`, the `Event Collector`, the `Diagnosis Component`, and the `Planner`. Several external components interact with the main application: the `Ticker` thread defines when the adaptation logic is triggered; the `Monitor` is a centralized collector of events; the `DLV-complex` reasoner is used during diagnosis and planning; the `H2 Database` assists diagnosis; and a `Reconfiguration` component enacts new variants.

The MDRC cycle starts with the `Ticker` thread triggering the `Event Collector`.

The `Event Collector` fetches not-yet-processed events from the `Monitor` thread, which is deployed to receive events from context sensors and agents. The `Event Collector` activates then the `Diagnosis` component, which determines failures with the aid of the `DLV-complex` tool and identifies which plans should be compensated on the basis of the information in the `H2 database`. As diagnosis is completed, the `User Interface` is updated to visually show the current status of the system on the requirements models. If failures are identified, the `Diagnosis` component activates the `Planner`, which generates applicable variants in the current context using `DLV-complex` as a planner. Finally, the `Planner` creates a `Reconfiguration` thread, which selects and enact one variant by commanding context actuators and interacting with agents.

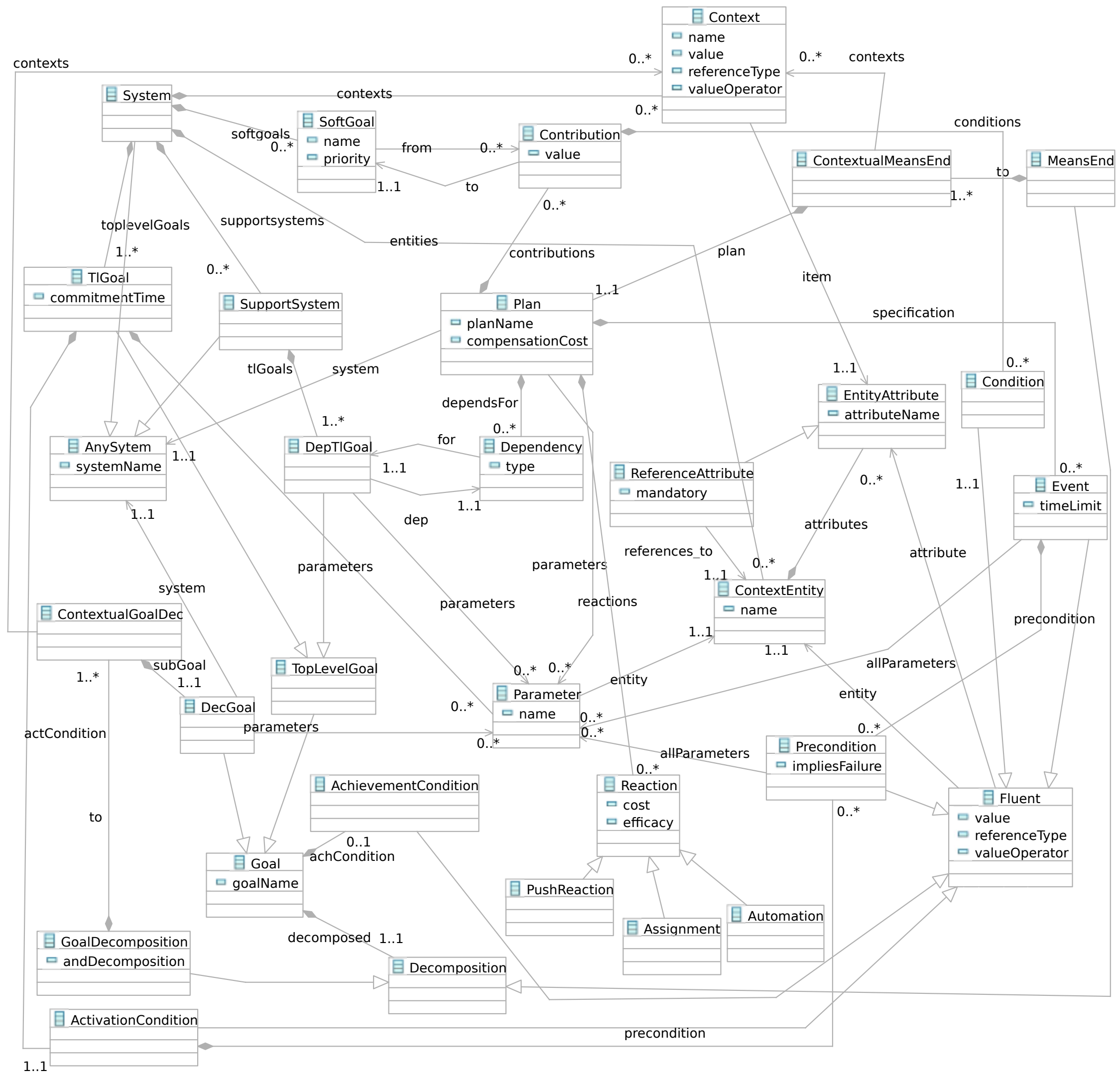


Figure 6.2: Meta-model for contextual goal models, the context model, and plan specifications

The requirements models for our prototype are specified according to the meta-models in Figure 6.2 and Figure 6.3. Figure 6.2 shows the meta-model for contextual goal models, the context model, and plan specifications. Figure 6.3 outlines the meta-model for domain assumptions. The most significant features of these meta-models are the following:

– goals are parametric and top-level goals have an activation condition. This way, multiple instances of the same goal are allowed. For example, goal Wake up might be activated at 8 am every day for each supported patient living in the smart-home;

– plans are linked to soft-goals by contextual contribution links. Contextual conditions are specified over the context model. Depending on the current context, the contribution of a plan to a certain soft-goal varies;

– goals can be declarative—i.e. they can have an achievement condition. A declarative goal is achieved only if the state of the world expressed by its achievement condition is met, regardless of the correct satisfaction of its children. A non-declarative goal is met if its sub-goals are met or, if means-end decomposed, an adequate plan is correctly carried out;

– the context model consists of a set of named contextual entities, each having a set of attributes. Attributes can either be simple (e.g. the temperature in a room is a real number) or refer to another entity (e.g. the default doctor of a patient);

– AND/OR-decompositions and means-end relations are contextual, and the contextual conditions are expressed over the context model;

– the two types of domain assumptions presented in Section 4.3 are supported. They differ for their consequent type: (i) a state of the world that should hold or (ii) an event that should happen within a certain time limit.

The meta-models we presented contains OCL constraints. The class diagram provides the coarse-grained structuring of requirements models, while OCL constraints restrict the allowed syntax. These constraints enable to check the well-formedness of a model. We illustrate this via examples.

The first constraint we present applies to class `Goal`. It says that goal instances should have a different name, i.e. the goal name is an unique identifier for the `Goal` class. To verify this, the OCL constraint takes all instances of class `Goal`, and checks that there are not different goals having the same value for attribute `goalName`.

```
context Goal inv:
not Goal.allInstances()
   -> exists(a: Goal,b: Goal | a<>b and a.goalName = b.goalName)
```

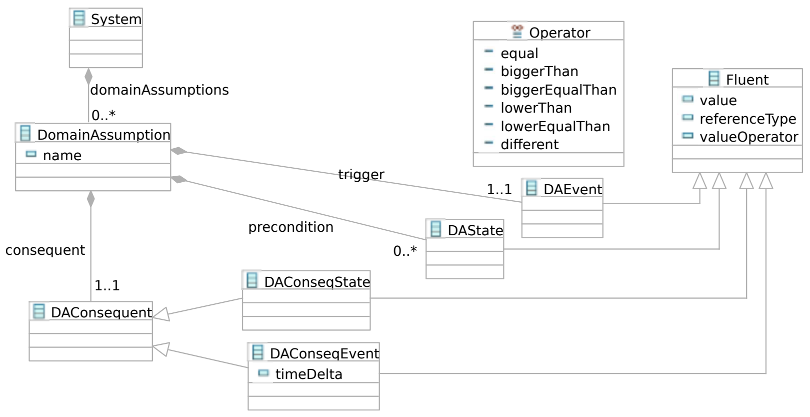


Figure 6.3: Meta-model for domain assumptions

The second constraint applies to class `Contribution`. It says that the attribute `value` of that class should be (i) greater or equal than minus one and (ii) less or equal that plus one. This constraint ensures that the value of contributions is in-between full negative (-1) and full positive (+1).

```
context Contribution inv:
value>=-1.0 and value <=1.0
```

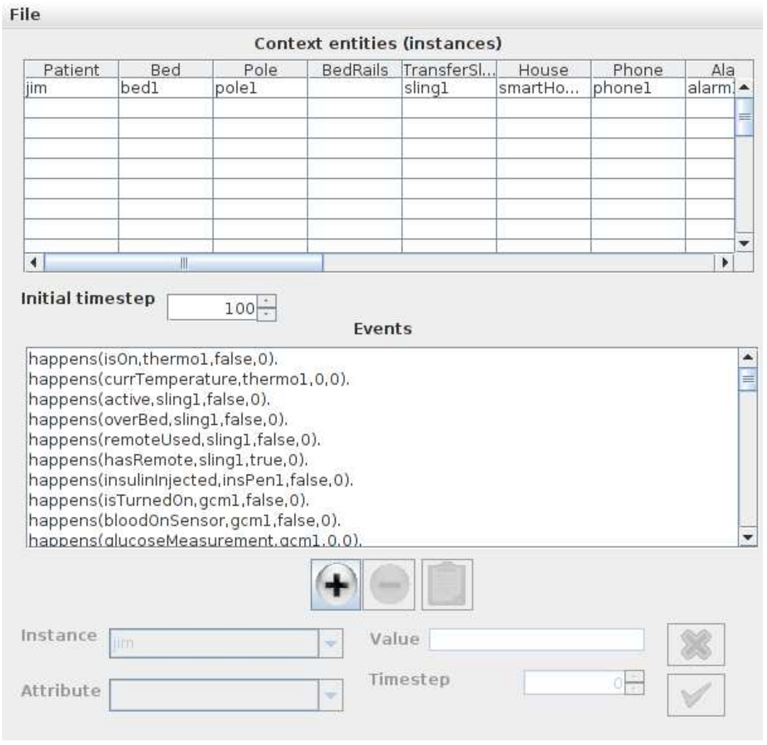
The third invariant applies to class `ContextualGoalDec`, and is more complex than the previous constraints. It ensures that contextual conditions in a goal decomposition variation point are expressed on entities that can be bound to parameters of the parent goal. For example, if an AND-decomposed goal Have breakfast has parameters patient and smart-home, then no contextual condition in the AND-decomposition should refer to a fire-fighter entity. The OCL constraint differentiates between root goals and non-root goals for technical reasons, but the two branches of the "if" express the same constraint. Specifically, each branch says that either one of the goal parameters is the referenced entity, or the entity is referenced by another entity in the contextual condition.

```
context ContextualGoalDec inv:
contexts->forAll(x | let y : ContextEntity = x.oclAsType(Context).item.entity in
  if from.parent.oclIsTypeOf(TlGoal) then
    (from.parent.oclAsType(TlGoal).parameters->exists(z |
      z.oclAsType(Parameter).entity=y)
      or contexts->exists(w | w.oclAsType(Context).referenceType=y.name))
  else if from.parent.oclIsTypeOf(DecGoal) then
      (from.parent.oclAsType(DecGoal).parameters->exists(z |
       z.oclAsType(Parameter).entity=y)
       or contexts->exists(w | w.oclAsType(Context).referenceType=y.name))
    else false
    endif
  endif)
```

The fourth invariant applies to class `Contribution`. It says that the contextual conditions of a contribution (if any exists) should refer to entities that belong to the parameters of the plan from which the contribution starts. Like the previous constraint, this is a well-formedness constraint that guarantees that contextual conditions in the contextual goal model refer to entities that can be correctly associated to the specific goal instance at runtime. For example, if a contextual contribution is dependent on the temperature of a room, but the decomposed goal has no parameter associated to a "room" entity, it will not be possible to guess which room should be considered.

```
context Contribution inv:
self.conditions->forAll(x | let y:ContextEntity = x.oclAsType(Condition).entity in
  self.contribFrom.parameters->exists(z |
    z.oclAsType(Parameter).entity=y))
```

Our prototype includes two tools for supporting designers. They allow for defining requirements models and simulation traces, respectively:



Patient	Bed	Pole	BedRails	TransferSl...	House	Phone	Ala
jim	bed1	pole1		sling1	smartHo...	phone1	alarm

Initial timestep 100

Events

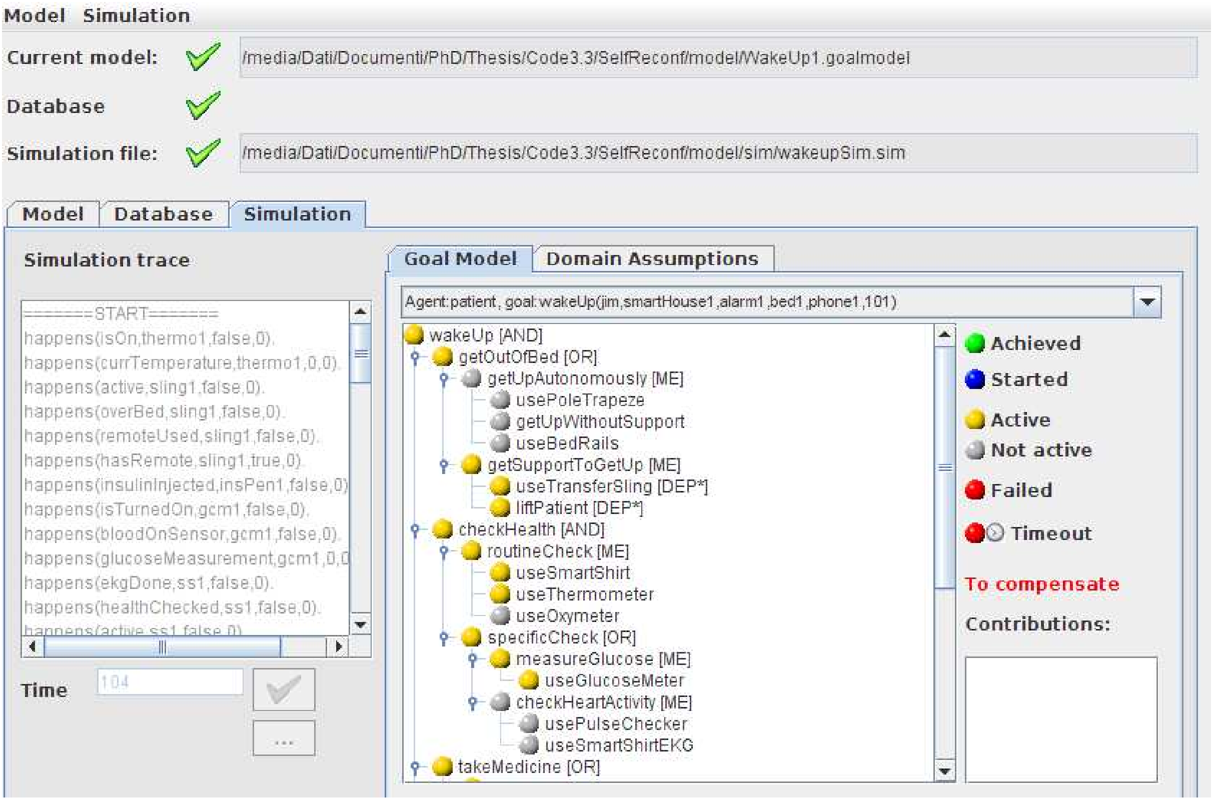
```
happens(isOn,thermo1,false,0).
happens(currTemperature,thermo1,0,0).
happens(active,sling1,false,0).
happens(overBed,sling1,false,0).
happens(remoteUsed,sling1,false,0).
happens(hasRemote,sling1,true,0).
happens(insulinInjected,insPen1,false,0).
happens(isTurnedOn,gcm1,false,0).
happens(bloodOnSensor,gcm1,false,0).
happens(glucoseMeasurement,gcm1,0,0).
```

Instance jim
Value
Attribute
Timestep 0

Figure 6.4: Screen-shot showing the simulation editor embedded in the prototype

– *Requirements models editor*: we use Java Emitter Templates (as in [Dam07]) to transform the EMF meta-model we presented—which is stored in an *ecore* file—into a requirements models editor deployable as an Eclipse application. The resulting application enables designers to define requirements models, and includes automated validation to check the model against the well-formedness rules defined by the meta-model class diagram and the OCL constraints.

– *Simulation editor*: we developed a simulation editor that enables designers to define simulation traces. Figure 6.4 shows a screen-shot of the editor. The upper part shows a matrix where instances of the contextual entities can be defined; contextual entity classes are taken from the context model specified using the requirements models editor. The designer can define at which timestep the simulation starts. The lower part of the editor enables to define the actual simulation, i.e. the events that occur. The events refer to the entity instances defined in the same interface.



Current model: ✓ /media/Dati/Documenti/PhD/Thesis/Code3.3/SelfReconf/model/WakeUp1.goalmodel

Database ✓

Simulation file: ✓ /media/Dati/Documenti/PhD/Thesis/Code3.3/SelfReconf/model/sim/wakeupSim.sim

Simulation trace

```
=======START=======
happens(isOn,thermo1,false,0).
happens(currTemperature,thermo1,0,0).
happens(active,sling1,false,0).
happens(overBed,sling1,false,0).
happens(remoteUsed,sling1,false,0).
happens(hasRemote,sling1,true,0).
happens(insulinInjected,insPen1,false,0)
happens(isTurnedOn,gcm1,false,0).
happens(bloodOnSensor,gcm1,false,0).
happens(glucoseMeasurement,gcm1,0,0
happens(ekgDone,ss1,false,0).
happens(healthChecked,ss1,false,0).
happens(active,ss1,false,0).
```

Time 104

Agent:patient, goal:wakeUp(jim,smartHouse1,alarm1,bed1,phone1,101)

```
wakeUp [AND]
  getOutOfBed [OR]
    getUpAutonomously [ME]
      usePoleTrapeze
      getUpWithoutSupport
      useBedRails
    getSupportToGetUp [ME]
      useTransferSling [DEP*]
      liftPatient [DEP*]
  checkHealth [AND]
    routineCheck [ME]
      useSmartShirt
      useThermometer
      useOxymeter
    specificCheck [OR]
      measureGlucose [ME]
        useGlucoseMeter
      checkHeartActivity [ME]
        usePulseChecker
        useSmartShirtEKG
  takeMedicine [OR]
```

● Achieved
● Started
● Active
● Not active
● Failed
● Timeout

To compensate

Contributions:

Figure 6.5: Runtime screen-shot of the prototype for settings with contextual variability

At runtime, requirements models are extensively used by our prototype. Before the architecture starts, a model is loaded and is automatically translated to the DLV-complex input format. Requirements models in the *ecore* file are expressed at the class-level. The translation to datalog supports instances, namely real agents and goal instances that constitute the application at runtime. In addition to requirements models, the architecture takes as input a simulation trace—possibly empty—and events received from the running application (agents and sensors). Currently, input from agents and sensors is read via socket listener. Figure 6.5 shows a screen-shot of the prototype in execution. The left side shows the simulation trace. The right side shows the current requirements and their status. Every root goal instance is shown as a tree; its nodes are coloured to denote the state of the goal (achieved, started, active, not active, failed, timeout). The status of domain assumptions is visualized in a different panel.

6.2 Prototype for settings with social variability

The second prototype we developed applies our architecture to open socio-technical systems. As argued earlier in this thesis (e.g. in Section 3.4), this class of systems is characterized by social variability: participating sub-systems vary, as well as the social relationships between them.

The prototype enables a sub-system (an agent) to adapt in response to threats and failures by switching from the current variant to a new one. Interaction plays a fundamental role in the notion of variant—that introduced by Definition 3.16—we employ here. The main features of our prototype are the following:

– Social relations are explicitly represented in terms of social commitments. The prototype takes into account the current commitments, and their status, to determine if agent adaptation is required and to identify valid variants to achieve the current goals. Commitments, being social relationships, originate from agent interaction. The prototype does not resolve exchanged messages into their meaning in terms of commitments. We assume that the agent is deployed in an appropriate middleware that performs this mapping.

– The focus is on variant generation and selection. While in the first prototype we implemented and tested the entire MDRC cycle, this second prototype investigate adaptation triggers (when to adapt) and the choice of the best variant. The distinctive feature of the prototype is to provide adaptation via goals and commitments.

– Efficient variant generation is provided by using a SAT-based automated reasoner $\mathcal{EL}^+$2SAT [SV09]. In turn, $\mathcal{EL}^+$2SAT includes the MiniSat2 SAT-solver [ES04]. Variant generation is reduced to propositional formulae. Then, $\mathcal{EL}^+$2SAT generates and return variants. The mapping from goal models to the $\mathcal{EL}^+$2SAT input format is automatically performed by our prototype, so that designers just have to specify goal models graphically.

- The prototype executes in an interactive live mode. The designer can add relevant events by interacting with the GUI. For example, the designer can add new commitments, notify changes in the state of the world, change the status of capabilities and commitments. The interactive live mode complements a static mode in which the prototype can be used offline to generate variants.
- An embedded policy editor enables the definition of fine-grained adaptation policies. The policy editor permits to define which are the triggers that stimulate an adaptation, as well as the variant generation and selection strategy. This way, designers can test the efficacy of different adaptation policies, and choose the one that seems the most adequate to the application domain.

We show now how our prototype exploits the $\mathcal{EL}^{+}$2SAT automated reasoning tool (Sebastiani and Vescovi [SV09]) to generate variants given an agent's goal model, a goal query, and a set of commitments. $\mathcal{EL}^{+}$2SAT was originally developed for concept subsumption and axiom pinpointing in the $\mathcal{EL}^{+}$ Description Logic. Given a theory $\mathcal{T}$ expressed as an ontology and given two concepts A and B in $\mathcal{T}$, $\mathcal{EL}^{+}$2SAT verifies if the interpretation of A is a subset of the interpretation of B in every model of $\mathcal{T}$. If so, it returns all minimal sets of axioms in $\mathcal{T}$—i.e. subsets of the original ontology—where A is subsumed by B.

Here, $\mathcal{EL}^{+}$2SAT is applied to a different problem, that of variant generation. Consequently, its operation has been slightly modified:

- the theory used by the tool is a propositional encoding (M) of the considered agent's goal model and the commitments that currently hold. Such encoding is based on the notion of goal support that we introduced in Section 3.4.4 in the definition of variant;
- we support a generic AND/OR query q over symbols in the theory.

The tool identifies minimal sets of axioms that satisfy q in the theory expressed by M, picking the axioms from the set S. The set S limits the search to those axioms that represent capabilities the agent can use, commitments the agent can take or make, and states of the world that already hold.

Figure 6.6 exemplifies the input files needed by $\mathcal{EL}^{+}$2SAT to perform variant generation. These files are automatically generated by our prototype and feed the automated reasoner. The models file M—part (a) in the figure—is specified in the DIMACS CNF format[3]. The first half of the file contains comments that ease the reading of the CNF

[3]www.satlib.org/Benchmarks/SAT/satformat.ps

encoding. In particular, it shows the mapping between propositional variables in the file and their meaning in requirements models. Here, variable 1 corresponds to commitment c_1; in particular, its meaning is that agent ag_1 can take such commitment from jim to support goal g_1. Variable 4 tells that the state of the world corresponding to g_5 holds. Variables 7 to 13 represent goals in the goal model. The query file—part (b) in the

```
c Agent name: ag1                         p cnf 13 11
c #1  <=> c1=C(jim,ag1,T,g1) --> g1       -1 7 0
c #3  <=> c3=C(ag1,ron,g1,g6) --> g1      -2 -8 9 0
c #4  <=> holds(g5)                       -3 7 0
c #5  <=> cap(g5)                         -4 8 0
c #6  <=> cap(g7)                         -5 8 0
c #7  <=> g1                              -6 10 0
c #8  <=> g5                              -11 -9 7 0
c #9  <=> g3                              11 -12 0
c #10 <=> g7                              11 -8 0
c #11 <=> g2                              9 -13 0
c #12 <=> g4                              9 -10 0
c #13 <=> g6
```

(a) Models file M

```
p cnf 13 1
-11 9 0
```

(b) Query file q

```
1
2
3
4
5
6
```

(c) Variables file S

Figure 6.6: $\mathcal{EL}^{+}$2SAT input files created by our prototype during variant generation

figure—represents the query in the DIMACS CNF format. The variables in this file are associated to the same entities as in the model file. Due to technical reasons, the query is expressed negated ($\neg q$). The actual query in the figure is $g_2 \wedge g_3$, and is represented as $\neg g_2 \vee \neg g_3$. The third file—part (c) in the figure—tells $\mathcal{EL}^{+}$2SAT which are the variables it should pick to define the minimal sets of axioms satisfying the query. These variables correspond to commitments, states of the world that hold, and capabilities.

Figure 6.7 provides some details about how $\mathcal{EL}^{+}$2SAT internally works. There are three main components: two SAT solvers (`Enumerator` and `T-Solver`) and a `Minimizer`. The `Enumerator` is in charge of selecting a subset of the variables in the file S. It does that by defining truth assignments for the variables in S, so that the truth assignment is consistent with a theory φ. Initially, φ is set to true. The truth assignment μ_i generated by the `Enumerator` becomes then input for the `T-Solver`, which checks whether there is a solution for $M \wedge \neg q \wedge \mu_i$. If this theory is satisfiable, then it means that the negation of the query can be satisfied, therefore the solution μ_i should be discarded—and indeed the negation of μ_i is added to the theory φ the enumerator handles. If the theory is unsatisfiable, the `T-Solver` identifies the conflict ψ_i, which is actually a solution for the considered query. Such conflict is given to the `Minimizer` to determine whether there are redundant variables that can be removed and that still preserve unsatisfiability. The minimal solution ψ_j is returned by the tool; then, the theory φ is updated by adding $\neg\psi_j$,

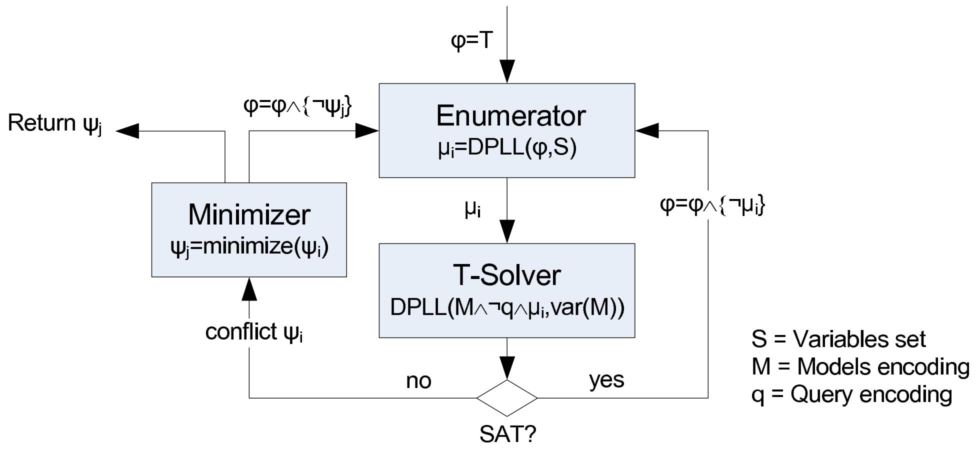


Figure 6.7: Basic operation of $\mathcal{EL}^+$2SAT [SV09] applied to variant generation

since such solution has already been found. The cycle iterates as long as the `Enumerator` finds new truth assignments. Details about $\mathcal{EL}^+$2SAT can be found in [SV09].

Algorithm 6.1 Encoding an agent to CNF

```
ENCODE(Agent $ag$, Query $q$, GoalModel $gm$, Commitment [] $c$, SoW [] $s$)
 1  Goal [] $roots \leftarrow$ GETRELEVANTROOTS($gm$, GETSYMBOLS($q$))
 2  GoalModel $gmp \leftarrow$ PRUNENOTRELEVANT($gm, roots$)
 3  if CHECKQUERY($q$)
 4    then return true
 5    else $idx \leftarrow 1$
 6         for each $comm \in c$
 7         do $idx \leftarrow$ ENCODECOMMITMENT($comm, ag, idx$)
 8         for each $sw \in s$
 9         do ADDCLAUSE($s, idx$)
10            $idx$++
11         for each $g \in$ GETGOALS($gmp$)
12         do if ISCAPABLEOF($ag, g$)
13              then ADDCLAUSE($g, idx$)
14                   $idx$++
15         ENCODEGOALMODEL($gmp$)
```

Let's take a closer look at the way the encoding to CNF is performed. Algorithm 6.1 describes the ENCODE algorithm, which takes in input an agent, a query, the agent's goal model, the current commitments, and the states of the world that currently hold. The first two lines reduce the goal model to preserve only those goal trees that are scoped with respect to the query (see Definition 3.15). To do this, the algorithm takes those goal tree roots that are ancestors of symbols in the query, and removes those trees in the goal model that are not scoped. In line 3 the algorithm checks if the query already holds; if so, no new variant is needed and the algorithm can return. Otherwise the encoding to CNF starts.

Variable idx, initialized in line 5, represents the propositional variable number associated to the capabilities, commitments, and states of the world (those in the file containing the variables set S). Commitments are encoded by the ENCODECOMMITMENT function (Algorithm 6.2). Each state of the world corresponds to a new clause and leads to the increment of idx. A variant can rely on states of the world that already hold. The encoding of goals (lines 11-14) differs only for the fact that capabilities are added. Finally, the goal model structure is encoded (line 15), following the mapping in Table 6.1.

Algorithm 6.2 Encoding commitments to CNF

```
ENCODECOMMITMENT(Commitment $c$, Agent $ag$, int $idx$)
 1  $\{p_1, \ldots, p_n\} \leftarrow$ TOCNFANDSPLIT($c.antecedent$)
 2  if $c.debtor = ag$
 3    then for each $p_i \in \{p_1, \ldots, p_n\}$
 4           do if not HASNONSCOPEDGOALS($p_i$)
 5                then ADDCLAUSE($q_i, idx$)
 6                     $idx$++
 7  if $c.creditor = ag$ or $ag.useThirdPartyCommitments$
 8    then $\{q_1, \ldots, q_l\} \leftarrow$ TOCNFANDSPLIT($c.consequent$)
 9         for each $q_i \in \{q_1, \ldots, q_l\}$
10         do if not HASNONSCOPEDGOALS($q_i$)
11                then ADDCLAUSE($c.antecedent \rightarrow q_i, idx$)
12                     $idx$++
13  return $idx$
```

The function ENCODECOMMITMENT (Algorithm 6.2) encodes a commitment to CNF, if relevant to the agent. Its input is a commitment, the agent under consideration, and the variables index used for the encoding. The variables index is returned to the caller function at the end of the algorithm. First, the function converts the antecedent to CNF and splits the conjuncts (line 1). The remainder of the algorithms depends on the role of the agent in the commitment:

- The branch of lines 3-6 is followed if the agent is a debtor. This reflects the rationale of Clause 2c in Definition 3.16. Each conjunct in the antecedent that does not contain non-scoped goals is encoded as a clause and idx is incremented. Each antecedent conjunct is a state of the world that the agent can realistically expect to obtain by making the commitment to some other agent (which will bring about the antecedent to detach the commitment).

- The branch of lines 7-12 is followed if the agent is creditor or is aware of an existing commitment that supports its goals. It reflects the rationale of Clause 2b in Definition 3.16. This branch considers third-party commitments only if the agent's adaptation policy specifies so. The algorithm converts to CNF the consequent of the commitment, then it split its conjuncts. For each conjunct that does not contain non-scoped goals, the algorithm adds an implication clause. The implication is from the antecedent of the commitment to a consequent conjunct. The AddClause function converts such clause to multiple CNF clauses.

Construct	Mapped	Mapping description
AND-decomposed$(g, \{g_1, \ldots, g_n\})$	✓	$g_1 \wedge \ldots \wedge g_n \rightarrow g$
OR-decomposed$(g, \{g_1, \ldots, g_n\})$	✓	$g_1 \vee \ldots \vee g_n \rightarrow g$
pos-contrib(g_1, g_2)	✓	$g_1 \rightarrow g_2$
neg-contrib(g_1, g_2)	×	post-processing

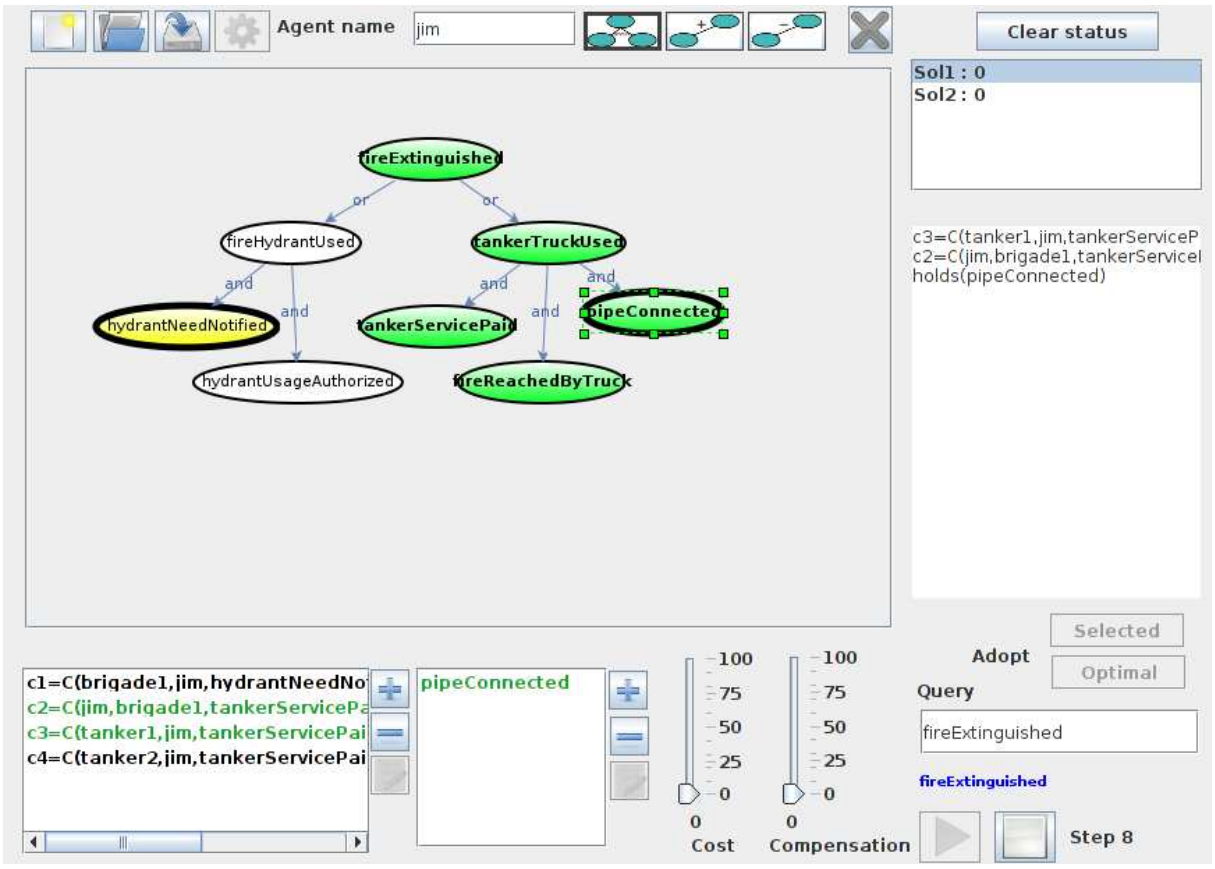
Table 6.1: Encoding a goal model to propositional logic

Table 6.1 shows how the different constructs in a goal model are mapped to propositional logic. This mapping is necessary to create the input files for $\mathcal{EL}^+$2SAT. Each AND-decomposition is mapped to an implication from the conjunction of the sub-goals to the parent goal. Each OR-decomposition is mapped to an implication from the disjunction of the sub-goals to the parent goal. These two rules reflect the semantics of goal support for goal decompositions. Positive contribution is mapped to an implication from the contributing goal to the contributed goal. Such mapping encodes the meaning of complete positive contribution. Negative contributions cannot be directly mapped to propositional logic due to the type of input $\mathcal{EL}^+$2SAT expects. Indeed, the tool works with Horn clauses, in which at least one literal should be positive. The CNF encoding of a negative contribution from g_1 to g_2 is $\neg g_1 \vee \neg g_2$; both literals are negative. To overcome such limitation, our tool performs the post-processing described in Algorithm 5.5.

Figure 6.8 shows a runtime screen-shot of the prototype. On the top-left side are placed the buttons to create new goal models, open/save files, and to configure the considered agent's adaptation policy. On the top-middle side there is a field containing the agent name and the palette to graphically draw the goal model (decomposition, positive contribution, negative contribution, element deletion). The main part of the GUI is occupied by the goal model of the agent. In order to let a designer define such model, we have exploited the JGraphX API[4], a Java library that allows for drawing and interacting with graphs. Below the goal model, from left to right, are placed (i) the set of current commitments; (ii) the states of the world that currently hold; (iii) two spinners that let the designer define cost and compensation cost for the selected goal.

The prototype deals with different goal and commitment states. The user can interactively update their state. Also, the user can add new commitments and add/remove

[4]http://www.jgraph.com/jgraph.html



Sol1 : 0
Sol2 : 0

c3=C(tanker1,jim,tankerServiceP
c2=C(jim,brigade1,tankerServiceI
holds(pipeConnected)

c1=C(brigade1,jim,hydrantNeedNo
c2=C(jim,brigade1,tankerServicePa
c3=C(tanker1,jim,tankerServicePai
c4=C(tanker2,jim,tankerServicePai

pipeConnected

fireExtinguished

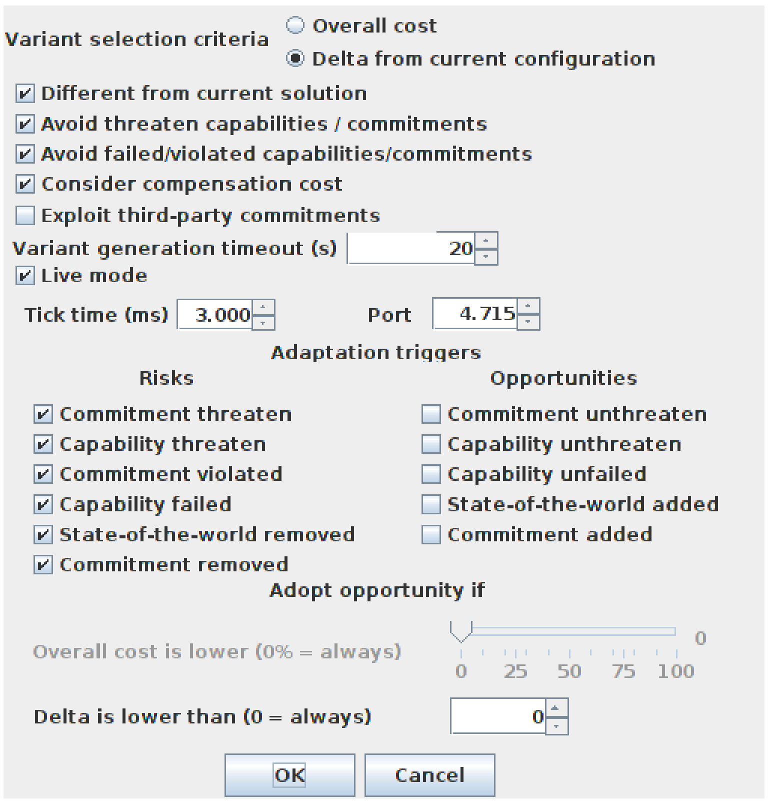
fireExtinguished

Figure 6.8: Runtime screen-shot of the prototype for settings with social variability

states of the world at runtime. The supported goal states are (i) *idle*: no specific status is set; (ii) *active*: the capability for that goal is being exploited; (iii) *supported*: the goal is supported by the current variant; (iv) *threatened*: there is some evidence that the goal is at risk; (v) *failed*: the current strategy for the achievement of the goal did not succeed. The supported commitment states are (i) *idle*: the commitment exists but is not being used by the agent in this variant; (ii) *active*: the commitment is part of the current variant; (iii) *threatened*: there is some evidence that the commitment is at risk; (iv) *violated*: the commitment is not correctly fulfilled. The *threatened* state of commitments represents the agent's view on a commitment. It represents the belief of the agent that the commitment is at risk. Another agent might have an opposite view on the same commitment. The different states are visually represented using intuitive colours (green for active, yellow for threatened, red for failed).

On the right side of Figure 6.8 are shown the current query (bottom-right) and the generated set of variants (above the query). Also, there are two buttons to start and stop the live interactive mode. The query can be expressed as an AND/OR formula, using binary AND and OR constructs wrapped by parentheses. So, for instance, query $g_1 \wedge g_2 \wedge g_3$ is expressed as "((g1 AND g2) AND g3)"; query $g_1 \vee (g_2 \wedge g_3)$ is expressed as "((g1 OR (g2 AND g3))". The generated variants are listed on the top-right side; the selected variant in the list is shown below the variants list. Variants are expressed as a set of commitments, capabilities, and states of the world. Our prototype exploits a notion of variant that extends that in Definition 3.16: a goal is supported if the corresponding state of the world holds. Such extension is useful for runtime adaptation, where the agent needs not put any effort to achieve a state of the world that already holds.

Figure 6.9 presents a screen-shot of the configuration options supported by our prototype. Many of these options define the agent's adaptation policy. We already provided details about variant selection policies in Section 5.2.2. We focus here on those configuration options that we did not describe there.


Variant selection criteria ○ Overall cost ● Delta from current configuration

☑ Different from current solution
☑ Avoid threaten capabilities / commitments
☑ Avoid failed/violated capabilities/commitments
☑ Consider compensation cost
☐ Exploit third-party commitments

Variant generation timeout (s) 20
☑ Live mode

Tick time (ms) 3.000 Port 4.715

Adaptation triggers

Risks	Opportunities
☑ Commitment threaten	☐ Commitment unthreaten
☑ Capability threaten	☐ Capability unthreaten
☑ Commitment violated	☐ Capability unfailed
☑ Capability failed	☐ State-of-the-world added
☑ State-of-the-world removed	☐ Commitment added
☑ Commitment removed	

Adopt opportunity if

Overall cost is lower (0% = always) 0

Delta is lower than (0 = always) 0

Figure 6.9: Adaptation policy editor

– *Variant generation timeout* ensures that $\mathcal{EL}^{+}$2SAT terminates within a fixed amount of time. The allowed syntax for goal models and commitments makes the variant generation problem an exponential problem. Consequently, an exhaustive variant generation procedure might not terminate within reasonable time. We will report on the performance of variant generation in Chapter 7.

- *Live mode* defines how the prototype should operate. Two options are available: live mode and passive mode. In live mode, the prototype cyclically monitors for changes—interactively created by the user—and responds to threats via adaptation. In passive mode, the designer can use the tool to generate variants and make offline experimentations. Live mode can be tuned by specifying the time tick, i.e. how frequently the agent should analyse monitored events, determine whether an adaptation is needed, and possibly perform an adaptation.
- *Adaptation triggers* specify why adaptation is required. The designer can choose which event types lead to adaptation. These triggers capture all events of a certain type, i.e. they do not refer to a specific event instance (e.g. commitment from agent x is violated, capability for goal g failed, etc.). Currently, the prototype supports two families of triggers:
  - *Risks* are events that endanger the current variant: an active commitment is threatened, an exploited capability is threatened, a commitment is violated, a capability fails, a state of the world does not hold any more, a commitment is cancelled by the debtor. Designers can activate individual risk types by ticking the corresponding check-box.
  - *Opportunities* are events that can potentially lead to better achievement of the agent's goals: a threatened commitment/capability is now not threatened any more, a capability is marked as unfailed (the capability is now available again), a state of the world is added, a new commitment is created.

6.3 Chapter summary

In this chapter we have described two prototype implementations of our architecture. The first prototype implements a complete Monitor-Diagnose-Reconcile-Compensate cycle and focuses on the impact of contextual factors on requirements. It implements the diagnosis algorithms of Section 5.1.1 and Section 5.1.2, as well as the reconfiguration algorithm—based on soft-goals and stability—of Section 5.2.1. The second prototype is suited for socio-technical systems with social variability and focuses on the identification of alternative variants based on capabilities and commitments (Section 5.2.2).

The principal contribution of this chapter is to demonstrate the feasibility of our conceptual architecture proposed in Chapter 4. Additionally, we implemented the diagnosis and reconfiguration algorithms devised in Chapter 5. The two prototypes show complementary aspects of our framework for self-adaptivity.

Chapter 7

Evaluation and scalability

The purpose of this chapter is to evaluate our approach. Since our framework spans from conceptual models to prototype implementations, evaluation should necessarily assess the efficacy of each component. To this end, we follow an evaluation methodology that relies extensively on case studies. The methodology consists of five steps:

1. define evaluation objectives. Which are the claims the evaluation should confirm or disconfirm?

2. select and describe a socio-technical system. The adequacy of the STS used as case study is justified with respect to the evaluation objectives;

3. model a relevant part of the STS. To demonstrate the applicability of the modelling primitives, they are used to represent a part of the STS in which variability is affected by contextual or social factors. This step involves applying the modelling framework presented in Chapter 3;

4. conduct simulations on self-adaptation. The models created in the previous step are used by our prototypes (proposed in Chapter 6) to perform simulations about self-adaptation. The purpose of this step is assessing the capability of our architecture (proposed in Chapter 4) to cope with failures and under-performance;

5. run scalability experiments. To determine if the prototypes can deal with larger models, scalability tests are performed. The analysis focuses on the time spent in diagnosis and reconfiguration. This step allows for assessing the efficacy of the algorithms presented in Chapter 5 and implemented in the prototypes described in Chapter 6.

We perform these steps twice. Each time we consider a specific evaluation objective:

O1. *Verify how well our framework supports STSs with contextual variability.* We use a case study about a smart-home for health support, in which the architecture supports a patient's goals. In particular, we apply our contextual goal modelling (Section 3.3.3) to model the case study, and the prototype of Section 6.1 to conduct simulations and assess scalability. Detailed in Section 7.1.

O2. *Verify how well our framework supports STSs with social variability.* We use an emergency response STS that focuses on dealing with hazardous materials. We examine this STS from the perspective of an incident commander who wants to assemble a response team. In particular, we apply the modelling framework based on goals and commitments presented in Section 3.4.2, and the prototype of Section 6.2 for simulations and scalability analysis. Detailed in Section 7.2.

7.1 Case study 1: smart-home for health support

A smart-home is an environment which provides assisted living with the help of technology. Research in this area is very active and interdisciplinary (see, e.g. Harper [Har03]). A smart-home is an STS, as we have shown in Chapter 1. We consider here a particular type of smart-homes for supporting elderly or handicapped people.

Our scenario is a variant of the "smart items" case study [CCG$^+$06] of the EU-funded Serenity project: a patient lives in a smart-home and is part of a socio-technical system supporting the patient in everyday activities (such as eating, sleeping, taking medicine, being entertained, consulting with doctor). Of particular importance is the health of the patient, which should be monitored and guaranteed by the smart-home. Moreover, a smart-home should minimize obtrusiveness. The patient's life experience should be affected as little as possible. The smart-home is expected to act unobtrusively, and should take more evident actions only if absolutely necessary.

The smart-home is equipped with Ambient Intelligence (AmI) devices that gather data—such as the patient's vital signs and the temperature in the bedroom—and enacts compensation actions—such as opening the door or alerting the medical centre if the patient feels giddy. Smart-homes are manifest exemplars of pervasive computing scenarios, where multiple devices are used both for monitoring and for enacting changes to the environment. For example, the prototype smart-home of the Serenity project exploits many sensors: a pulse oxymeter to gather patient's heart rate and saturation level, a set of cameras capable of motion detection and object tracking, a wireless sensor network to authenticate doctors and social workers, magnetic fields to detect open doors and windows. The equipment in the house (lights, doors, windows, heating, and so on) are connected by the KNX communication bus[1]. In addition, the house includes a number of actuators for the entrance door, the windows and the blinds to enable automatic opening and closing. Moreover, lights and temperature can be adjusted automatically; some pieces of equipment can be raised or lowered, etc.

7.1.1 Modelling contextual variability

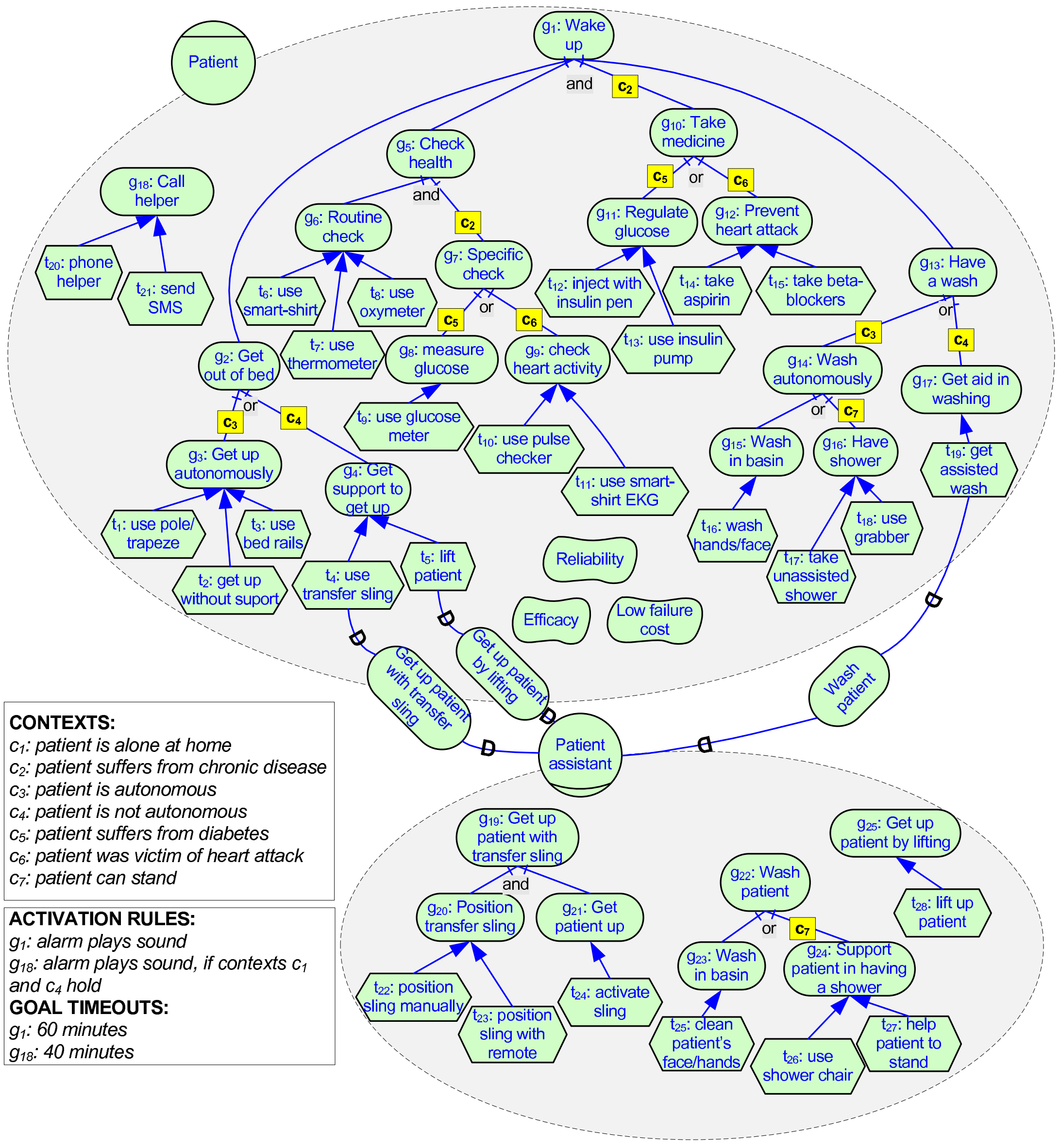
A smart-home is an STS with contextual variability. Many contextual changes are relevant for the goals of the home inhabitant and influence his current goals and the way these goals are achieved. For example, there could be no food in the fridge, the temperature in the kitchen might be too high, some electric appliance (oven, fridge, . . .) might break, the patient might forget or lose the home keys, he might be constricted in bed by a flu, social workers enter the house to perform regular tasks such as bringing medicines, etc.

We detail part of the case study where the smart-home system helps the patient to wake up. We have applied the contextual goal modelling process we outlined in Section 3.3.5. In this section, we show the main outcomes of such application. Specifically, we present the contextual goal models that represent the requirements of the smart-home system. To ease reading, we provide textual descriptions for the contexts, rather than showing the complete context analysis. Later in the section, we show task specifications where pre- and post-conditions are expressed on the contextual model derived from context analysis.

Figure 7.1 presents the goals of a patient using a contextual goal model (see Section 3.3.3 and Section 4.3.1). The top-level goal g_1: Wake up is AND-decomposed to sub-goals g_2: Get out of bed, g_5: Check health, g_{10}: Take medicine, and g_{13}: Have a wash. The decomposition of g_1 to g_{10} is labelled c_2: the achievement of g_1 requires the achievement of the sub-goal g_{10} only if the context c_2 (patient suffers chronic disease) is valid. g_2 is OR-decomposed to sub-goals g_3: Get up autonomously and g_4: Get support to get up, which are valid alternatives if the patient is autonomous and not autonomous, respectively. Three plans achieve g_3: the patient can use a bed pole (or a bed trapeze) to get up more easily (t_1), get up without support (t_2), or use bed rails (t_3).

To achieve goal g_4, possible means are t_4: use transfer sling and t_5: lift patient. Each of these plans originates a dependency on actor *Patient assistant*, for goal Get patient up with transfer sling and Get patient up by lifting, respectively. Notice that the goals of the assistant are less specific than those of the patient; indeed, these are general assumptions and do not characterize in detail the plan an assistant will carry out. The first dependum (the object of a dependency) becomes top-level goal g_{19} of the patient assistant. g_{19} is AND-decomposed to g_{20}: Position transfer sling and g_{21}: Get patient up. g_{20} is means-end decomposed to plans t_{22} and t_{23}: the former indicates manual positioning of the sling, the

[1]a standard bus that allows devices in smart-homes to communicate: http://www.knx.org/



CONTEXTS:
c_1: patient is alone at home
c_2: patient suffers from chronic disease
c_3: patient is autonomous
c_4: patient is not autonomous
c_5: patient suffers from diabetes
c_6: patient was victim of heart attack
c_7: patient can stand

ACTIVATION RULES:
g_1: alarm plays sound
g_{18}: alarm plays sound, if contexts c_1 and c_4 hold
GOAL TIMEOUTS:
g_1: 60 minutes
g_{18}: 40 minutes

Figure 7.1: Contextual goal model for a smart-home patient: goals wake up and call helper

latter refers to the use of a remote control to position it. Plan t_{24}: Activate sling is the only available alternative to achieve g_{21}. The dependency for goal Get patient up by lifting results in patient assistant's top-level goal g_{25}, which is means-end decomposed to t_{28}: lift up patient. The patient's goal g_5: Check health is AND-decomposed to sub-goals g_6: Routine check and g_7: Specific check; the latter goal should be achieved only if the patient suffers from chronic diseases (in context c_2). Possible means to achieve g_6 are using a smart shirt to check the health (t_6), using a thermometer (t_7), or using an oxymeter (t_8). g_7 is OR-decomposed to sub-goals g_8: measure glucose and g_9: check hearth activity; each decomposition link is subject to a different context: g_8 is possible if the patient suffers from diabetes, g_9 should be achieved if the patient was victim of heart attack. The only plan to achieve g_8 is t_9: use glucose meter. g_9 can be achieved either by using a pulse checker (t_{10}) or by using a smart shirt with EKG capabilities (t_{11}). The trees for the other sub-goals of g_1 are similar, thus we don't detail them here. The patient has also another top-level goal, g_{18}: Call helper. This goal can be achieved by calling a helper by phone (t_{20}) or sending an SMS (t_{21}).

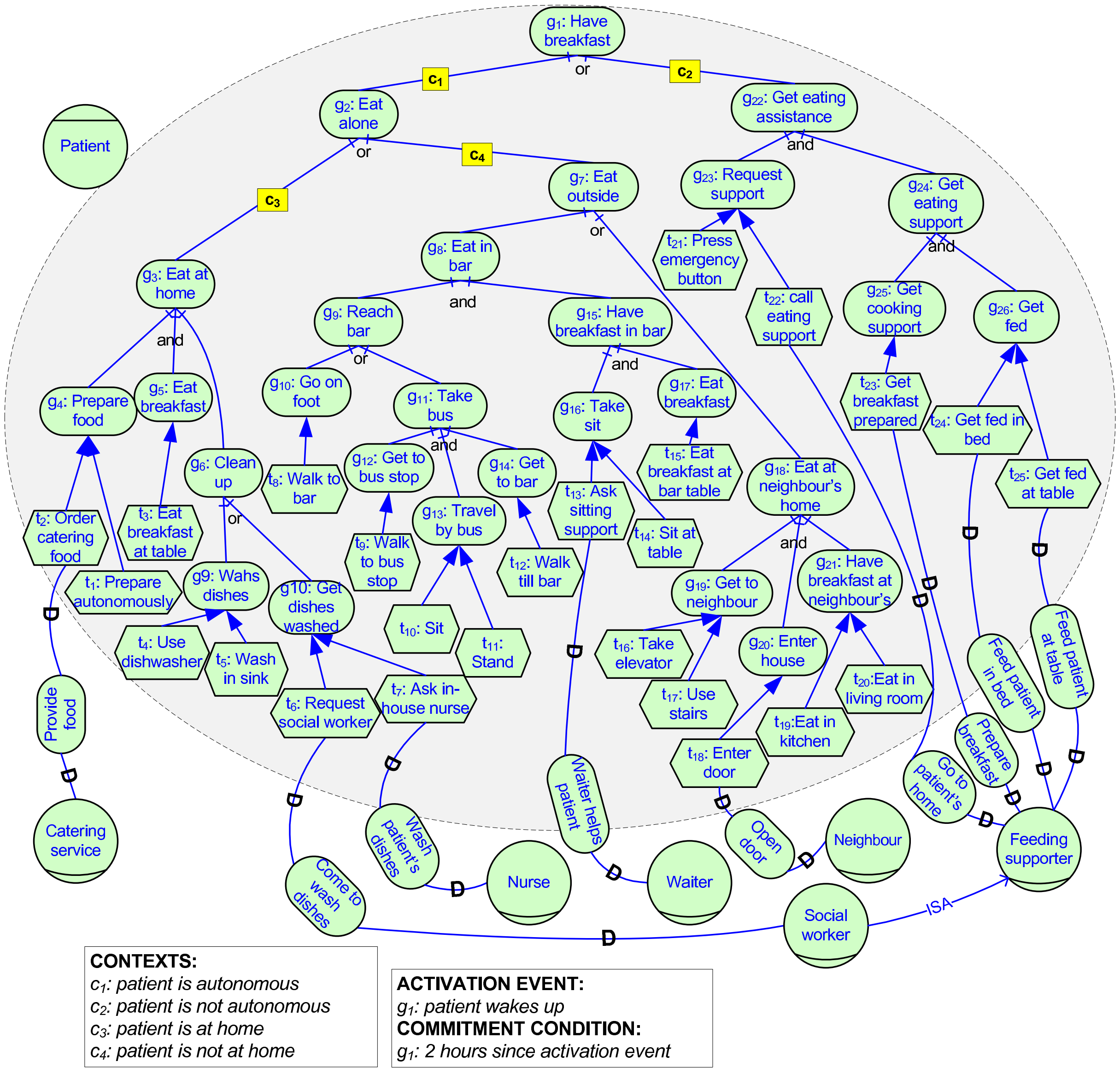


Figure 7.2: Contextual goal model for a smart-home patient: goal have breakfast

Figure 7.2 shows a different part of the scenario which describes goal g1: Have breakfast. g_1 is activated when the patient wakes up (activation event). The patient should internally commit to achieve g_1 within two hours since goal activation. Four different contexts characterize the scenario: in $c1$ the patient is autonomous, in c_2 the patient is not autonomous, in c_3 the patient is at home, in $c4$ the patient is not at home. If the patient is autonomous (c_1 holds) g_1 is decomposed into the sub-tree of goal g_2: Eat alone; if c_2 holds g_1 is decomposed into the sub-tree of goal g_{22}: Get eating assistance. In the former case, c_3 activates the sub-tree of goal g_3: Eat at home, whereas c_4 activates the sub-tree of goal g_7: Eat outside. When eating at home, the patient has to prepare food (g_4), eat breakfast (g_5), and clean up (g_6). Goal g_4 is means-end decomposed to two alternative tasks: t_1: Prepare autonomously and t_2: Order catering food. The latter task requires interaction with the external actor Catering service, which should fulfil goal Provide food for the successful execution of t_2. The other sub-trees of Figure 7.2 are structured in a similar way, thus we don't detail them here.

Plan name	**Precondition**	**Action**	**Time**
t_1: Use pole/trapeze	bed b has a pole pl	pl is touched	5
	patient *p* is in bed *b*, b has pole pl, pl is touched	p stands up	10
t_2: Get up without support	**patient *p* is in bed *b***	p stands up	10
t_3: Use bed rails	bed b has rails r	r are activated	7
	patient *p* is in bed *b*, b has rails r, r are activated	p stands up	10
t_6: Use smart shirt	patient p has a smart shirt s, ***s* is active**	s performs a health check	15
t_7: Use thermometer	patient p has thermometer t, ***t* is active**	t measures temp > 35°C	35
t_8: Use oxymeter	patient p has oxymeter o, ***o* is working**	o measures heart rate	16
	patient p has oxymeter o, ***o* is working**	o has measured saturation	16
t_9: Use glucose meter	patient p has glucose meter gm, ***gm* is turned on**	gm has blood on its sensor	28
	patient p has glucose meter gm, ***gm* is turned on**, ***gm* has blood on its sensor**	gm measures glucose level	30
t_{10}: Use pulse checker	patient p has pulse checker c	c measured pulse	25
t_{11}: Use smart shirt EKG	patient p has smart shirt s	s performs EKG	40
t_{12}: Inject with insulin pen	patient p has insulin pen i, p has glucose meter gm, ***gm* measured glucose level**	i injects insulin	50
t_{13}: Use insulin pump	patient p has insulin pump i, ***i* is under skin**	i pumped insulin	30
t_{20}: Phone helper to get up	house h has phone ph	ph is dialling	10
	house h has phone ph, ***ph* is not dialling**	ph called a helper	15
t_{24}: Activate sling	bed b has sling sl, ***sl* is active** and over bed	patient p stands up	45

Table 7.1: Specification for some plans in Figure 7.1. Preconditions in bold are critical

Table 7.1 shows a semi-formal specification for some of the patient's plans in Figure 7.1. Plans are expressed according to the plan specification introduced in Section 4.3.3. We explain now the semantics of the specification for some of these plans.

Plan t_9: Use glucose meter is defined by two actions: the glucose meter has blood on its sensor, and the glucose meter measures glucose level. The first action has a non-critical precondition (the glucose meter should belong to the considered patient), and a critical precondition (the glucose meter is turned on). If the glucose meter is turned off but blood is detected on its sensor, a plan failure is detected. The second action has an additional critical precondition: the glucose meter should have blood on its sensor. In general, our plan specification does not prescribe plan sequencing. However, sequencing can be explicitly expressed. Here, the event corresponding to the first action (blood on sensor) is used as a critical precondition for the second action, so that the second action is to execute after the first one. The timeout for the first action is 28, the timeout for the second action is 30. Plan t_{12}: Inject with insulin pump is specified by one action: the insulin pen injects insulin. The action has two non-critical preconditions: the insulin pen belongs to the patient and the patient has a glucose meter. It has a critical precondition too: the glucose meter has measured glucose level. This means that plan t_{12} should be carried out after plan t_9. The specification of plan t_{20}: Phone helper to get up consists of two actions. First, the phone should be dialling (time limit 10). The action precondition tells that the phone should belong to the smart-home. Second, the phone should call a helper. This action has a critical precondition: the phone should not be dialling. Therefore, if the sensors notify that a helper is called while the phone is still dialling, a failure is detected.

7.1.2 Simulations: adaptation in the smart-home STS

We have experimented our prototype for settings with contextual variability (Section 6.1) on the smart-home scenario described in the previous section. We describe some simulations concerning the contextual goal model in Figure 7.1. The goal model is composed of two agents, 25 goals, 28 plans, and 3 soft-goals. The context model consists of 18 entities having 67 attributes. Overall, 67 different event types are relevant to our architecture, such as "phone is dialling", "bed rails are active", "patient enters bathroom", "patient exits home". We have conducted our experiments on a machine equipped with an AMD Athlon(tm) 64 X2 Dual Core Processor 4200+ processor, 2GB RAM, and running Linux ubuntu 2.6.31-16-generic ♯53-Ubuntu SMP i686, Java OpenJDK Runtime Environment (IcedTea6 1.6.1) (6b16-1.6.1-3ubuntu1).

We tested the requirements model on several realistic scenarios. We show here three simulations that involve diagnosis and compensation. We express these simulations in terms of timesteps. In order to automate testing in a simulated environment, we developed a module that simulates the enactment of a chosen reconfiguration.

Simulation 1 *Marco* is a diabetic patient living in a smart-home. Currently he cannot stand up autonomously, for he is still weak after a bad flu. However, he can stand up if assisted. He is typically supported by a social worker named *Mike*. Marco is in bed at timestep 0, he is alone at home. At timestep 1 the alarm rings in his bedroom. Marco is supposed to call a helper and to wake up: goals g_{18} and g_1 are instantiated. At timestep 2 the phone dials; this gives evidence that Marco is calling a helper, maybe Mike. At timestep 5 an event says that a helper is called: the smart-home system receives a confirmation SMS from Mike. □

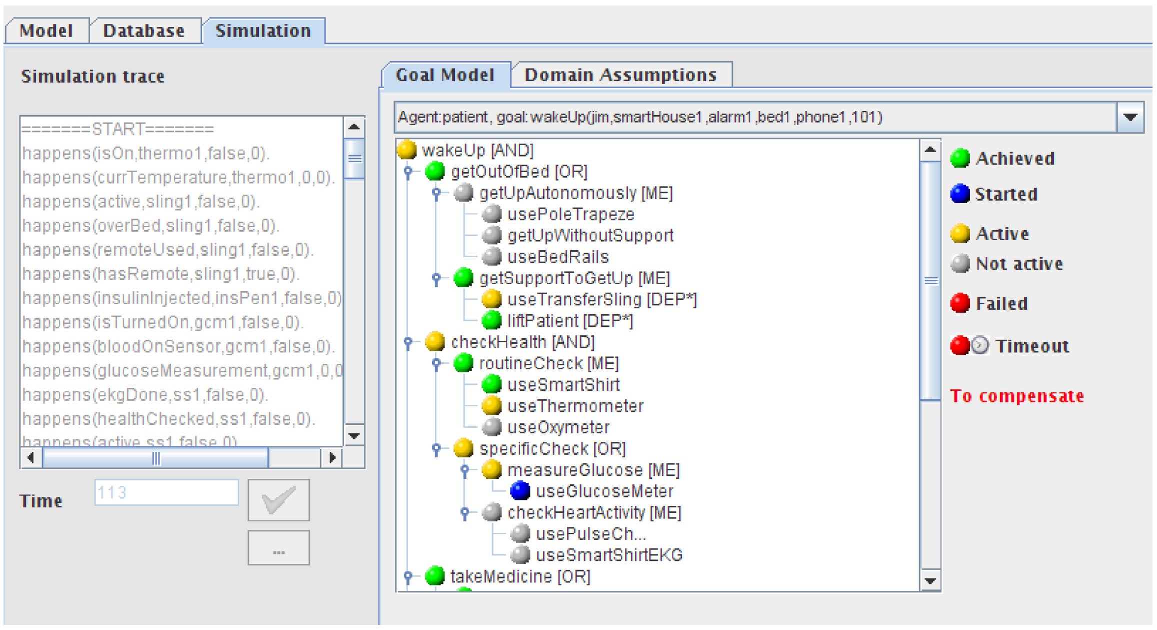
This simulation raises a failure, according to the specification for plan t_{20}. Indeed, the phone is still dialling (see Table 7.1) as the helper confirms he will come. This is a critical precondition, therefore the diagnosis component identifies this failure. The root cause for this failure is either a fault in the sensor that detects phone dialling status or in the sensor that detects received SMSs. No tolerance policy is specified, thus the architecture should compensate for this failure. Twelve alternatives are generated by DLV-complex, then the best alternative is chosen according to Algorithm 5.4 (in Section 5.2.1). The selected strategy supports both active goals: g_{18} and g_1. The failed plan t_{20} is marked to be compensated, and plan t_{21} (send SMS) is selected. The reconfiguration simulator takes care of enacting the chosen strategy and lets the agents achieve their own goals. Plan t_{21} is automated by sending an SMS to Mike. In a real environment, some compensation actions could fail—or have no effect—implying further reconfiguration.

Simulation 2 Marco's alarm rings at timestep 1. At timestep 2 his phone starts dialling. At timestep 3 the phone stops dialling and a helper is called. This time t_{20} is carried out correctly. At timestep 4 Marco is not alone in his house any more; moreover, the sling is over his bed. At time 9, Marco stands up with the support of a person. However, the sling has not been touched. □

Also in Simulation 2, the event sequence leads to a failure, specifically plan t_{24} of the assistant fails. Indeed, the patient is standing and the sling is over the bed, but the sling is not active. Clearly, the patient could hardly be standing up, for the sling is supposed to be just above Marco. This might have different interpretations: either the patient is standing and the sling activation sensor is broken, the patient standing sensor failed, or maybe the assistant is not even there. Regardless of the real cause, the architecture plans possible alternatives and selects the best one, which includes notifying the helper to lift the patient. It might be the case that the person in the house found a way to bypass using the sling; in any case, the system is just notifying such person, acting in an unobtrusive way. If Marco does not achieve his other goals, more obtrusive actions will be taken.

Simulation 3 After some days, Marco feels better and can stand up autonomously. At timestep 1 the alarm rings, and Marco gets up without support (t_2) at timestep 10. Now, he is expected to measure glucose level, a fundamental activity for diabetic patients (g_8 should be achieved). He should use the glucose meter and put a blood drop on the sensor. Let's consider two variations now: (i) at timestep 25, Marco uses his insulin pen and injects insulin; (ii) Marco doesn't carry out any further relevant action till time 32. □

This third simulation produces a failure detected by the architecture at timestep 25 (in the first variation) or at timestep 32 (in the second variation). The first failure refers to plan t_{12}, and is detected because the critical precondition (the glucose level has been measured) is violated. Indeed, Marco has not measured his glucose before injecting insulin. This is a dangerous situation, for injecting an incorrect amount of insulin might lead to hypoglycaemia and later to hyperglycaemia. Our architecture reacts to this failure by choosing plan t_{13} (use insulin pump) and compensating t_{12} by notifying the nurse assigned to the Marco. The prototype tries to enact plan t_{13} by sending a notification to Marco telling him to wear the insulin pump. The insulin pump performs continuous glucose level monitoring and injects insulin whenever needed. In the second variation, the failure is a timeout for plan t_9. There is no alternative to such plan, thus the architecture has to retry with the same plan. Also, plan failure is compensated by alerting the nurse.



```
=======START=======
happens(isOn,thermo1,false,0).
happens(currTemperature,thermo1,0,0).
happens(active,sling1,false,0).
happens(overBed,sling1,false,0).
happens(remoteUsed,sling1,false,0).
happens(hasRemote,sling1,true,0).
happens(insulinInjected,insPen1,false,0)
happens(isTurnedOn,gcm1,false,0).
happens(bloodOnSensor,gcm1,false,0).
happens(glucoseMeasurement,gcm1,0,0
happens(ekgDone,ss1,false,0).
happens(healthChecked,ss1,false,0).
happens(active,ss1,false,0).
```

Figure 7.3: Screen-shot of the architecture applied to the smart-home case study (Simulation 2)

Figure 7.3 shows a snapshot of the architecture running Simulation 2. The current simulation trace is shown on the left side, whereas requirements monitoring is on the right side. Goal models are represented as trees, the status of every goal is represented by a coloured circle. Plans are leaf-level nodes in goal trees. The status of domain assumptions are shown in a different tab (hidden in Figure 7.3). In Figure 7.3, the architecture has started its reaction to t_{24} failure: Marco has been lifted by the helper, the smart shirt has performed a routine check, the patient is using the glucose meter to measure his blood, and Marco has already taken his medicine.

These simulations demonstrate the capability of our first prototype to cope with different types of failures related to contextual factors. Specifically, we have shown how the architecture can detect plan failures and expired timeouts, and how it responds to these issues via different reconfigurations. In the first simulation, the plan is automated to ensure a helper is actually notified. The other two simulations minimize the obtrusiveness of the smart-home system: notifications are sent to specific people so that they can help the patient upon double checking the need of intervention. During our simulations, the prototype performed well during diagnosis and reconfiguration. On average, CPU usage was below 9%, with a maximum value of almost 67% and some other peaks, but also some values close to 0%. Memory (heap) usage follows a pattern where heap allocation (high peaks are ∼61MB) is immediately followed by heap deallocation (low peaks are ∼9MB). On average, heap usage is less than 33MB. For what concerns performance, we can conclude that the size of the models used for this case study is not critical for our prototype. On average, diagnosis took 863ms, with a maximum of 1215ms and a minimum of 787ms. Planning for alternatives and selecting the best reconfiguration took on average 203ms.

7.1.3 Scalability experiments for the first prototype

We have performed two types of scalability experiments to verify different phases of the MDRC cycle: (i) failure diagnosis, i.e. how long the architecture takes to determine failures (Algorithms 5.1 and 5.2); (ii) system reconfiguration, i.e. the time needed to derive alternatives (Algorithm 5.3) and to choose the best one (Algorithm 5.4).

We have verified diagnosis scalability on goal models of growing size. We have increased the size of goal models in three ways: (i) the number of top-level goals of an agent; (ii) the number of agents in the model; (iii) the depth of a goal tree. We report results concerning all these dimensions.

Table 7.2 summarizes scalability results obtained by increasing the number of top-level goals. We took a basic goal tree composed of 15 goals and we replicated it to obtain multiple goal trees, thus increasing the total number of goals. The six columns in the table represent number of goal trees, number of goals, number of datalog rules, diagnosis time, diagnosis time per goal, and diagnosis time per datalog rule, respectively. The results show that the tool scales very well till 180 goals (12 top-level goals), for the time per goal

Trees	Goals	Rules	Time	$\frac{Time}{Goals}$	$\frac{Time}{Rules}$
1	15	131	218	14.533	1.664
2	30	255	318	10.600	1.247
3	45	379	526	11.689	1.387
6	90	751	799	8.878	1.064
12	180	1495	2146	11.922	1.435
24	360	2983	7101	19.725	2.380
48	720	5959	25615	35.576	4.299

Table 7.2: Diagnosis scalability: increasing the number of top-level goals; time in ms

is always below 15ms. Larger goal models increase the time per goal till 35ms for 720 goals. However, this is still a good result since time does not grow exponentially.

Agents	Goals	Rules	Time	$\frac{Time}{Goals}$	$\frac{Time}{Rules}$
2	20	291	213	14.550	1.366
3	25	320	291	12.800	1.099
5	40	340	379	8.500	0.897
11	70	463	525	6.614	0.882
21	120	793	915	6.608	0.867
41	220	1964	1695	8.927	1.159
81	420	5870	3255	13.976	1.803
161	820	17831	6375	21.745	2.797

Table 7.3: Diagnosis scalability: increasing the number of agents; time in ms

Table 7.3 reports on scalability for requirements models with multiple agents. We don't change the supported agent, and make it depend on an increasing number of other agents. The agents acting as dependees are cloned, each one having one small goal tree composed of five goals. The table columns represent number of agents, number of goals, number of rules, diagnosis time, time per goal and time per rule. The diagnosis scales very well (linearly) till 420 goals (81 agents); the result is a bit worse with 820 goals (161 agents) but the growth is still not exponential.

Depth	Goals	Rules	Time	$\frac{Time}{Goals}$	$\frac{Time}{Rules}$
5	18	241	156	13.388	1.545
10	23	286	191	12.435	1.497
20	33	375	261	11.364	1.437
40	53	727	401	13.717	1.813
80	93	2076	681	22.323	3.048
160	173	7184	1241	41.526	5.789
320	333	30071	2361	90.333	12.737

Table 7.4: Diagnosis scalability: increasing goal model's depth; time in ms

Table 7.4 details scalability results about goal models with increasing depth. In order to increase depth, we generated goal decompositions with just one sub-goal. In this setting, the diagnosis mechanisms scale less well than in the other two experiments: good scalability is measured till 93 goals (depth 80). However, notice that we are measuring unrealistically deep goal models.

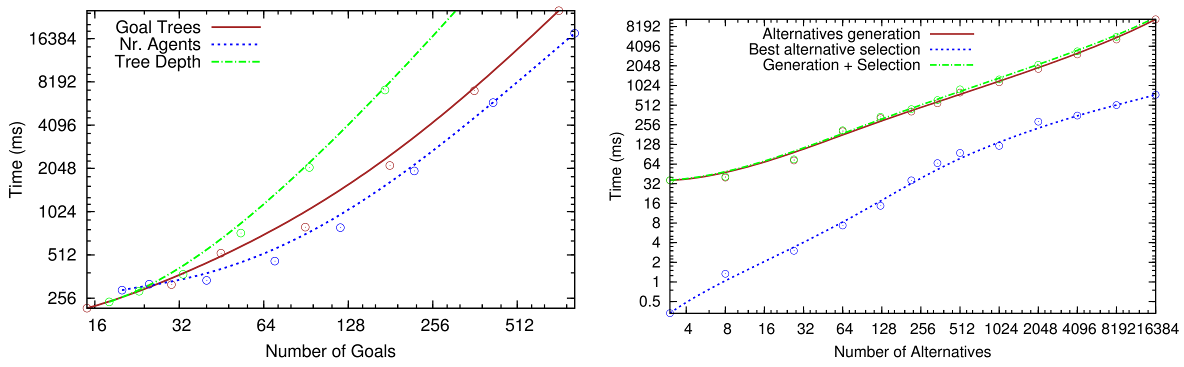


Figure 7.4: Scalability evaluation for diagnosis (left side) and reconfiguration (right side) mechanisms

The left-hand side of Figure 7.4 shows a chart plot summarizing diagnosis scalability. The chart is created using a logarithmic scale on both axes. The three plots represent the three conducted experiments: the chart shows that diagnosis works better with many small goal models (even with many agents) than with a single large goal model (i.e., increasing its depth). Overall, the diagnosis mechanisms perform well for medium-sized requirements models.

The basic unit to verify reconfiguration scalability is not the number of goals. Reconfiguration is the selection of an alternative variant to achieve the requirements of the system. Therefore, reconfiguration corresponds to (i) generating viable variants and (ii) selecting the best variant. Scalability has to be assessed with respect to the number of variants. To check reconfiguration scalability we created several goal models with increasing number of variants. In particular, we introduced additional options to variation points: more sub-goals in OR-decompositions and more plans in means-end decompositions.

Table 7.5 reports on scalability for reconfiguration mechanisms. The five columns represent number of variants, time taken to generate the variants, time taken to select the best variant, generation time per variant, and selection time per variant, respectively. Each experiment was repeated three times; the overall time is approximated to the millisecond in columns 2 and 3. Results show that the implemented reconfiguration mechanisms scale very well. The growth is linear with the number of variants, as can be seen in the ratio columns. Both generation and selection perform efficiently. Selection time is much smaller than generation time. The right-hand side of Figure 7.4 graphically resumes the scalability results concerning reconfiguration, also shows the overall reconfiguration time.

♯ var	Gen time	Sel time	$\frac{Gen\ time}{\sharp var}$	$\frac{Sel\ time}{\sharp var}$
3	36	<1	12.000	0.111
8	40	1	4.958	0.166
27	73	3	2.691	0.111
64	204	7	3.182	0.115
125	325	15	2.597	0.117
216	408	36	1.867	0.166
343	547	66	1.594	0.193
512	798	95	1.559	0.186
1024	1148	122	1.121	0.119
2048	1839	286	0.898	0.140
4096	3064	354	0.747	0.111
8192	5198	514	0.635	0.063
16384	10604	738	0.647	0.045

Table 7.5: Reconfiguration scalability: increasing the number of variants; time in ms

7.2 Case study 2: hazardous materials emergency response

Emergency response involves multiple social actors that interact—via a technical infrastructure—to reach the emergency location, determine the hazard severity, plan a proper response, and timely enact the response. Therefore, emergency response defines a socio-technical system. This case study is about emergencies where hazardous materials are involved. It is based on a publicly available document released by the South Dakota department of public safety[2]. Such document[3], titled "Hazardous Material Plan Workbook", defines general response procedures for emergency handling involving improper handling or accidental release of hazardous materials (HazMat).

When improperly handled or accidentally released, HazMats threaten life, property, and the environment. Quick and efficient response is mandatory to effectively manage and mitigate an incident. Inadequate response is very costly in terms of lives, money, and environmental damage. The procedure described by the document is complemented and refined by local emergency plans. The document covers all phases concerning emergency response: response preparedness, response, and recovery.

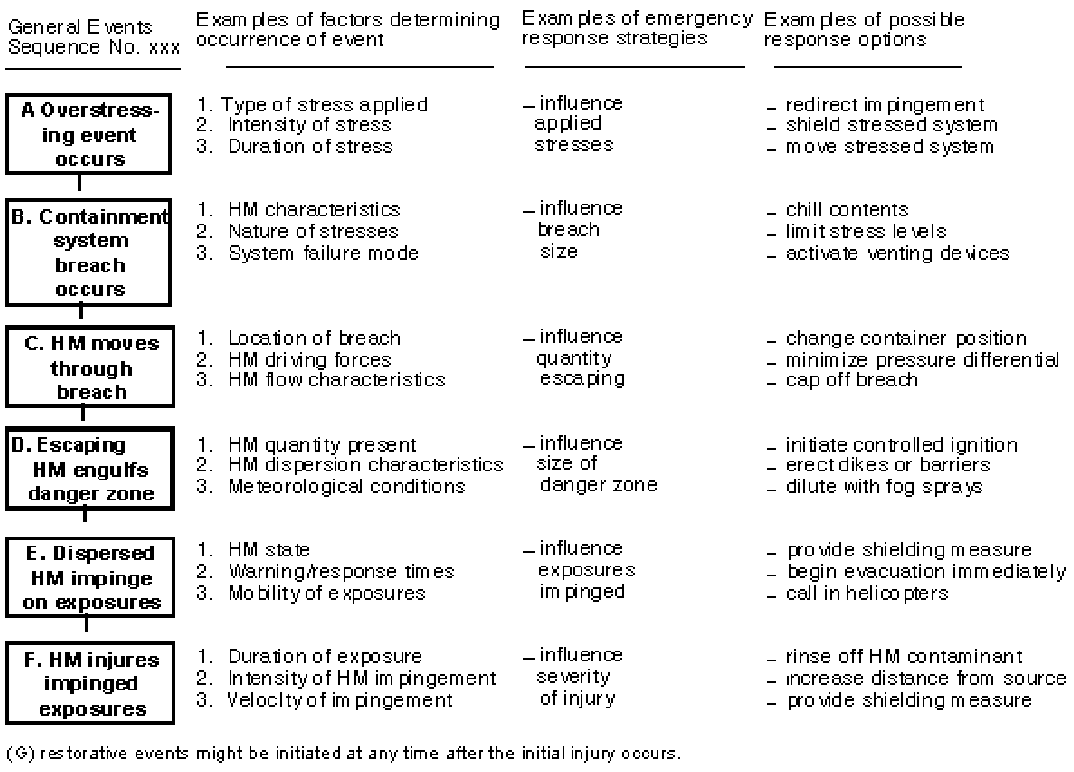
We focus here on *hazardous material response* phase (section C in the document). Several actors are involved in such phase. The *Local Emergency Planning Committee* (LEPC) of the county is the main authority, which is responsible for the correct management of HazMat threats. The *County dispatcher* serves as the 24 hour contact for notification of HazMat incidents, and has to alert the most adequate agencies to initiate response. The first public safety officer getting to the location becomes the *Incident commander*. He

[2]http://www.dps.sd.gov/emergency_services/emergency_management/

[3]http://dps.sd.gov/emergency_services/emergency_management/images/hazmat_plan_workbook.pdf

is authorized to nominate a *designee* that acts as vice-commander. *Response Personnel* is chosen and coordinated by the incident commander, and is responsible for enacting the command and control procedure. *Public safety officials* conduct off-site evacuation operations if requested. *Designated agencies* are alerted by the incident commander to effectively tackle the emergency actuating the most adequate response.

We complement the procedure description found in the document with information taken from the DECIDE procedure applied to HazMat emergencies [Ben75]. The DECIDE procedure acronym defines six steps: (i) Detect HazMat presence; (ii) Estimate likely harm without intervention; (iii) Choose response objectives; (iv) Identify action options; (v) Do best option; and (vi) Evaluate progress. In particular, we consider the emergency response options proposed by Benner [Ben75] and shown in Figure 7.5.



General Events Sequence No. xxx	Examples of factors determining occurrence of event	Examples of emergency response strategies	Examples of possible response options
A Overstressing event occurs	1. Type of stress applied 2. Intensity of stress 3. Duration of stress	– influence applied stresses	– redirect impingement – shield stressed system – move stressed system
B. Containment system breach occurs	1. HM characteristics 2. Nature of stresses 3. System failure mode	– influence breach size	– chill contents – limit stress levels – activate venting devices
C. HM moves through breach	1. Location of breach 2. HM driving forces 3. HM flow characteristics	– influence quantity escaping	– change container position – minimize pressure differential – cap off breach
D. Escaping HM engulfs danger zone	1. HM quantity present 2. HM dispersion characteristics 3. Meteorological conditions	– influence size of danger zone	– initiate controlled ignition – erect dikes or barriers – dilute with fog sprays
E. Dispersed HM impinge on exposures	1. HM state 2. Warning/response times 3. Mobility of exposures	– influence exposures impinged	– provide shielding measure – begin evacuation immediately – call in helicopters
F. HM injures impinged exposures	1. Duration of exposure 2. Intensity of HM impingement 3. Velocity of impingement	– influence severity of injury	– rinse off HM contaminant – increase distance from source – provide shielding measure

(G) restorative events might be initiated at any time after the initial injury occurs.

Figure 7.5: Hazardous materials emergency response options (from [Ben75])

7.2.1 Modelling social variability

HazMat response procedures are a social-intensive activity, where several social actors (humans, teams, organizations) perform a coordinated action that will ultimately lead to controlling the emergency. Social variability is an inherent property in this setting.

Alternative teams and individuals are available and commit to deliver various services. For example, the incident commander can typically choose among different people the most suitable designee; also, many emergency response teams can act as response personnel.

The social structure of an STS responding to an emergency varies over time. Agents involved in emergency response often have to adapt to ensure their objectives are achieved. This happens, for instance, if some agent breaks a previously made commitment. For example, a response team might be unable to successfully define a restricted area because team members are busy with other tasks. Then, the incident commander will have to find a replacement team. Sometimes, adaptation is required due to contextual changes. For example, the severity of the hazard might increase, and this might require to find an alternative response agency with specific skills.

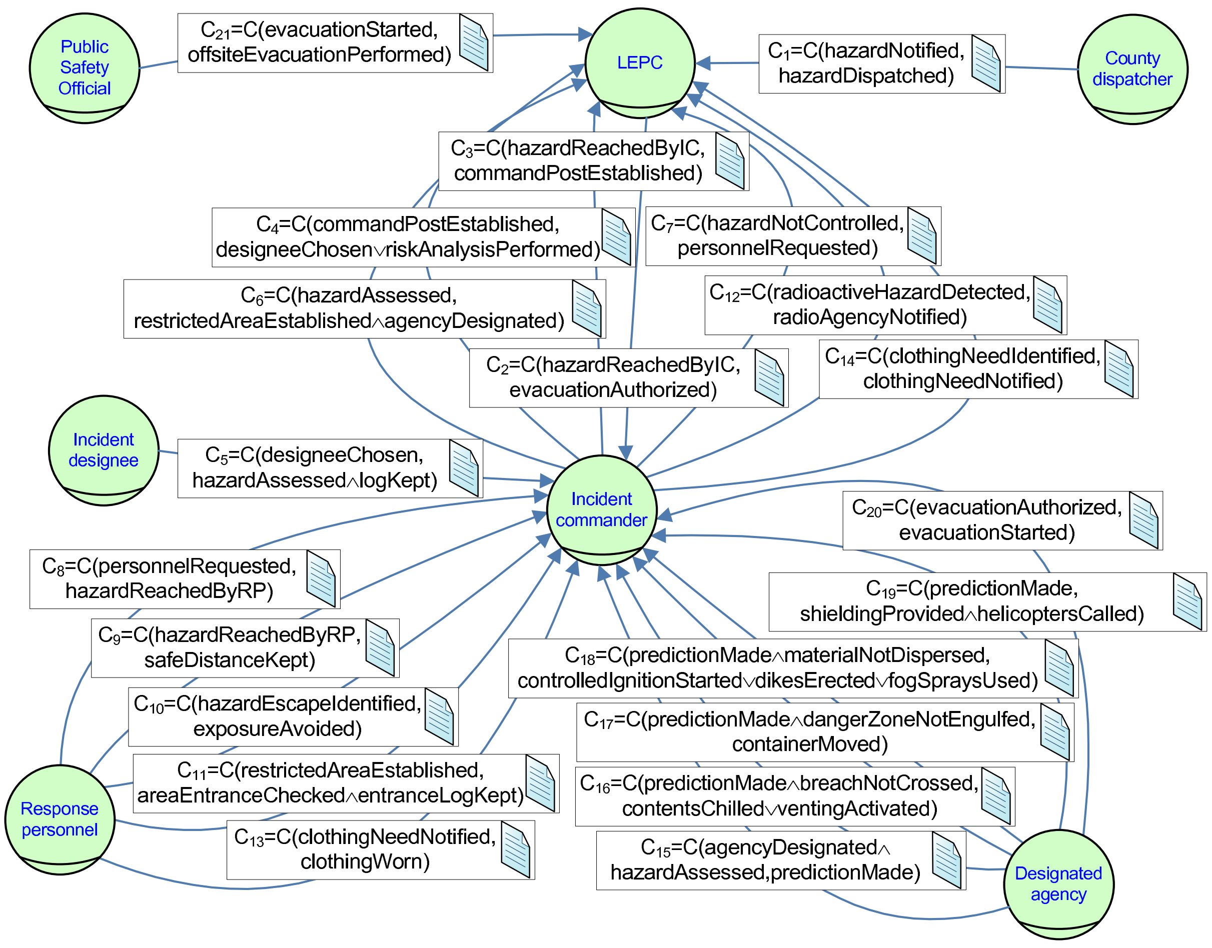

Public Safety Official
C21=C(evacuationStarted, offsiteEvacuationPerformed)
LEPC
C1=C(hazardNotified, hazardDispatched)
County dispatcher
C3=C(hazardReachedByIC, commandPostEstablished)
C4=C(commandPostEstablished, designeeChosen∨riskAnalysisPerformed)
C7=C(hazardNotControlled, personnelRequested)
C6=C(hazardAssessed, restrictedAreaEstablished∧agencyDesignated)
C12=C(radioactiveHazardDetected, radioAgencyNotified)
C2=C(hazardReachedByIC, evacuationAuthorized)
C14=C(clothingNeedIdentified, clothingNeedNotified)
Incident designee
C5=C(designeeChosen, hazardAssessed∧logKept)
Incident commander
C20=C(evacuationAuthorized, evacuationStarted)
C8=C(personnelRequested, hazardReachedByRP)
C19=C(predictionMade, shieldingProvided∧helicoptersCalled)
C9=C(hazardReachedByRP, safeDistanceKept)
C18=C(predictionMade∧materialNotDispersed, controlledIgnitionStarted∨dikesErected∨fogSpraysUsed)
C10=C(hazardEscapeIdentified, exposureAvoided)
C17=C(predictionMade∧dangerZoneNotEngulfed, containerMoved)
C11=C(restrictedAreaEstablished, areaEntranceChecked∧entranceLogKept)
C16=C(predictionMade∧breachNotCrossed, contentsChilled∨ventingActivated)
Response personnel
C13=C(clothingNeedNotified, clothingWorn)
C15=C(agencyDesignated∧ hazardAssessed,predictionMade)
Designated agency

Figure 7.6: Commitments in the HazMat case study

We apply the modelling framework of Section 3.4.2 and specify both the service-oriented application (in terms of a commitments protocol), and a particular agent that wants to participate in the application. In Figure 7.6 we show a possible commitments protocol for the HazMat case study. The figure shows the expected commitments between the involved roles. At runtime, specific agents play these roles and introduce social variability in the scenario. Figure 7.6 helps an agent designer to verify whether such agent will be able to achieve its goals by playing a certain role, provided that it finds agents whose commitments are compatible with those expressed at the role level.

The county dispatcher commits to the LEPC that, if a hazard is notified, the hazard will be dispatched to all agencies in the emergency notification roster (C_1). The LEPC commits to the incident commander that, if the commander reaches the hazard, evacuation will be authorized (C_2). This commitment reflects the rule that a public officer playing role incident commander can start an evacuation only if it receives an authorization. The incident commander makes a set of commitments to the LEPC: (C_3) as soon as he reaches the hazard, an on-scene command post with communication capabilities will be established outside the immediate danger area; (C_4) if the command post is established, he will either choose a designee or perform risk analysis by himself; (C_6) if the hazard ia assessed, a restricted area will be established and an emergency response agency will be designated; (C_7) if the hazard is not under control, additional personnel will be requested; (C_{12}) if a radioactive hazard is detected, a specialized agency will be notified; (C_{14}) if special clothing is needed, then the need of wearing special clothes will be notified to all involved rescuers. The incident designee commits to the incident commander that, if he is chosen as designee, the hazard will be assessed and an event log will be kept (C_5).

The public safety official commits to the LEPC that, if an evacuation is started, then offline evacuation will be performed. The response personnel makes a set of commitments to the incident commander: (C_8) if personnel is requested, then the hazard will be reached; (C_9) as the hazard is reached, the response personnel will be in charge for keeping safe distance between people and the hazard; (C_{10}) if a hazard escape is identified, then necessary measures to avoid exposure will be adopted; (C_{11}) if a restricted area is established, then the response personnel will check entrance to the area and will keep a log of people entering/exiting the area; (C_{13}) if the need to wear special clothing is notified, then such clothing will be worn.

The designated agency is responsible for responding to the emergency by controlling the HazMat and putting out possible fires. There are several commitments from the agency to the incident commander: (C_{15}) if the agency is designated and hazard severity is assessed, then a prediction about the evolution of the hazard will be made; (C_{16}) if the evolution prediction is made and the breach has not been crossed by the HazMat, then either the contents will be chilled or the venting system will be activated; (C_{17}) if the prediction is made and the danger zone is not engulfed by escaping HazMat, then the

HazMat container will be moved; (C_{18}) if the prediction is made and the material is not dispersed, either a controlled ignition will be started, or protective dikes will be erected, or fog sprays will be used; (C_{19}) if the prediction is made, shielding will be provided and helicopters will be called; (C_{20}) if evacuation is authorized, then the evacuation will be started. Commitments C_{16}-C_{19} represent different response strategies that cover events from B to E in Figure 7.5.

Figure 7.7 shows the specification of agent Mike (its goal model). Mike wants to adopt role incident commander in the HazMat scenario. At design-time, such goal model can be checked against the commitments in Figure 7.6. Mike has four top-level goals: responding a hazard (hazardResponded), tackling radioactive hazard (radioactiveHazardTackled), evacuating people when needed (peopleEvacuated), and ensuring body protection to rescuers (bodyProtecionEnsured).

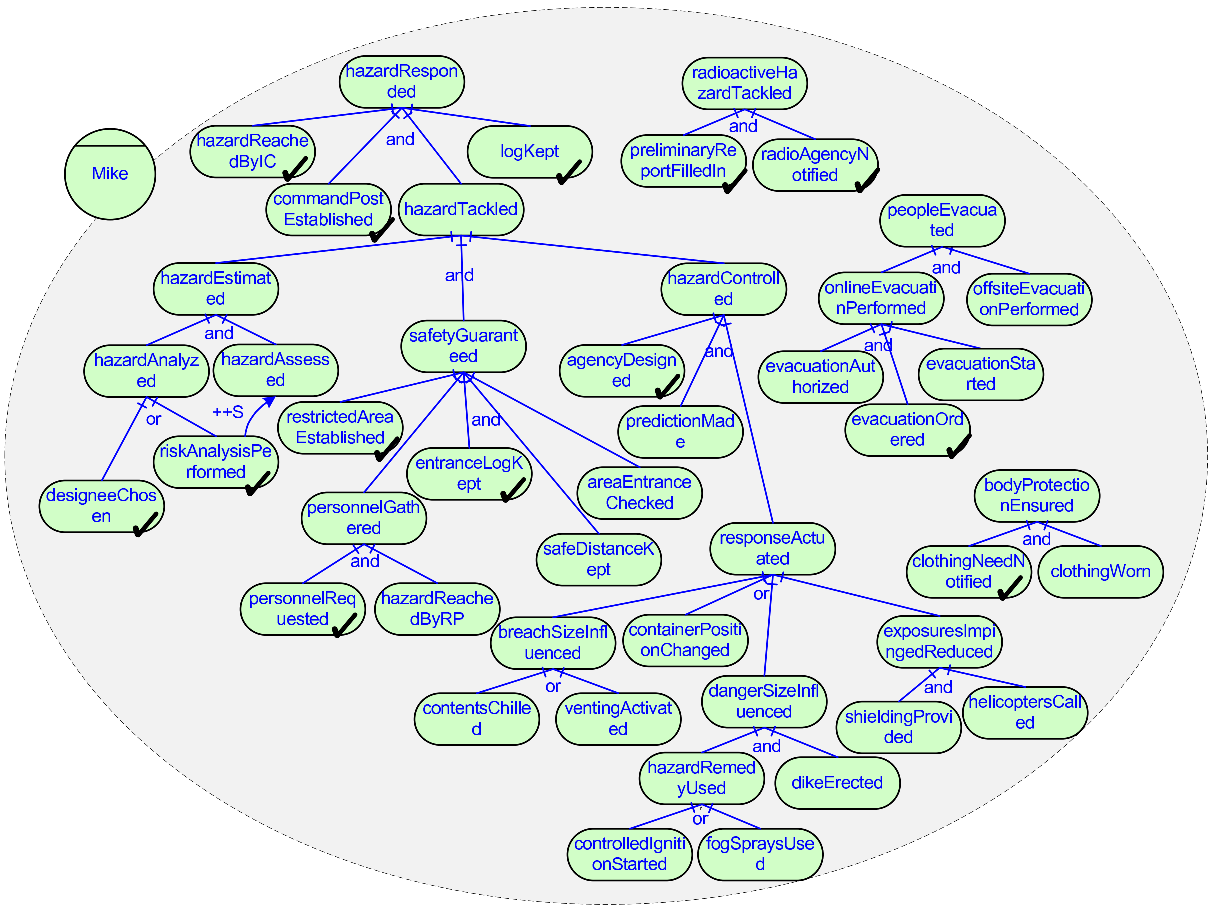


Figure 7.7: Goal model for agent Mike who wants to play Incident Commander

The first top-level goal is AND-decomposed to four sub-goals: the hazard should be reached by the incident commander, a command post should be established, the hazard should be tackled, and an event log should be kept. Mike is capable of hazardReached-ByIC, commandPostEstablished, and logKept. To achieve hazardTackled, Mike has to estimate the hazard, guarantee safety, and control the hazard. Goal hazardEstimated is AND-decomposed to analysing the hazard and assessing the hazard. The former sub-goal is achievable either by choosing a designee (designeeChosen) or performing risk analysis (riskAnalysisPerformed). Goal riskAnalysisPerformed contributes positively to (supports) hazardAssessed. To guarantee safety, Mike has to establish a restricted area, gather personnel, keep an entrance log, keep safe distance, and check area entrance. He is capable of restrictedAreaEstablished and entranceLogKept. To achieve personnelGathered, he has to request personnel (he is capable of this goal) and to ensure the hazard is reached by response personnel.

Achieving goal hazardControlled requires to designate a response agency, make a prediction about the hazard evolution, and actuate a response. Mike is capable of agencyDesignated. In order to achieve responseActuated, Mike has different alternatives: influencing the breach size, changing the position of the container, influencing the danger size, or reducing the impinged exposures. Goal breachSizeInfluenced can be achieved either by chilling the contents or activating the venting system. To achieve dangerSizeInfluenced, he has to use a remedy and to erect a protection dike. Goal hazardRemedyUsed is OR-decomposed into controlledIgnitionStarted and fogSpraysUsed. In order to reduce the impinged exposures, Mike has to provide a shielding and to call helicopters. Notice that Mike has no capability for responding to hazards. He has to rely on commitments made by other agents.

The second top-level goal, radioactiveHazardTackled, is AND-decomposed to filling in a preliminary report and notifying an agency that can deal with radioactive hazards. Mike is capable of both preliminaryReportFilledIn and radioAgencyNotified. To achieve the third top-level goal, peopleEvacuated, Mike has to perform both an online evacuation and an offline evacuation. To perform an online evacuation, he needs to authorize the evacuation, order the evacuation, and start it. He is capable only of ordering the evacuation. The fourth root goal, bodyProtectionEnsured, is AND-decomposed to notifying the need to wear special clothing and wearing such clothing. Mike is capable of the former sub-goal.

7.2.2 Simulations: adaptation in the HazMat STS

The purpose of this section is to show how our architecture—specifically, the prototype for STSs characterized by social variability (Section 6.2)—adds self-adaptive capabilities to agent Mike in the HazMat STS. The prototype can be used either at design-time to run simulations, or at run-time to support Mike in his decision-making. We do not examine the operationalization of adaptations. We limit our analysis to the generation of variants and the selection of the best one according to Mike's adaptation policy (we use the framework proposed in Section 5.2.2).

Before describing the simulations, we list the commitments from other agents that currently hold in the scenario (Table 7.6). In addition, we specify the activation and the compensation cost for each of these commitments, which are communicated by the debtor as the commitment is made. In Table 7.7 we specify activation and compensation cost for Mike's capabilities. The commitments in Table 7.6 are made by agents playing roles in the protocol specified by Figure 7.6. Due to their autonomy, the agents make different commitments from those specified by the roles they play.

Commitment	AC	CC
C_a=C(Designee1, Mike, designeeChosen, hazardAssessed ∧ logKept)	10	4
C_b=C(Lepc, Mike, hazardReachedByIC, evacuationAuthorized)	1	25
C_c=C(Rteam1, Mike, personnelRequested, hazardReachedByRP ∧ safeDistanceKept)	7	10
C_d=C(Rteam1, Mike, restrictedAreaEstablished, areaEntranceChecked ∧ entranceLogKept)	5	1
C_e=C(Rteam3, Mike, restrictedAreaEstablished ∧ hazardReachedByRP, areaEntranceChecked ∧ entranceLogKept)	6	2
C_f=C(Rteam1, Mike, clothingNeedNotified, clothingWorn)	1	1
C_g=C(Agency1, Mike, agencyDesignated ∧ hazardAssessed, predictionMade)	6	2
C_h=C(Agency2, Mike, agencyDesignated, hazardAssessed ∧ predictionMade)	7	0
C_i=C(Agency1, Mike, predictionMade ∧ dangerZoneNotEngulfed, containerPositionChanged)	5	3
C_j=C(Agency2, Mike, predictionMade ∧ materialNotDispersed, fogSpraysUsed ∧ dikeErected)	7	16
C_k=C(Agency3, Mike, predictionMade, shieldingProvided ∧ helicoptersCalled)	26	40

Table 7.6: Commitments in the hazard materials response simulation

Commitment C_a is made by agent Designee1 and tells that, if the designee is chosen, the hazard will be assessed and the log will be kept. The activation cost is 10, the compensation cost 4. C_a is a concrete version of commitment C_5 in the role-based diagram of Figure 7.6. Agent Lepc commits (C_b) to authorize evacuation if the hazard is reached by the incident commander; the compensation cost is much higher than the activation cost, since cancelling C_b includes a lot of work on behalf of agent Lepc. C_b is a concrete version of C_2. Agent Rteam1 commits to reach the hazard and keep safe distance if personnel is requested (C_c), check area entrance and keep an entrance log if a restricted area is established (C_d), and wear special clothing if such need is notified (C_f). Agent Rteam3 commits to check area entrance and keep entrance log if a restricted area is established and the hazard is reached by response personnel (C_e). In other words, Rteam3 will make this commitment only if some other rescue team is already at the emergency location. Commitments C_g-C_k are about controlling the hazard. Three agencies are involved. Agency1 commits to make a prediction if the agency is designated and the hazard is assessed (C_g); the activation cost is 6, the compensation cost is 2. Agency2 commits to make the prediction and assess the hazard, if the agency is designated (C_h); activation and compensation costs are 7 and 0, respectively. Agency1 commits to change the container position if the prediction is made and the danger zone is not engulfed (C_i). Both activation (5) and compensation (3) are cheap. Agency2 commits to use fog sprays and erect a dike if a prediction is made and material is not dispersed (C_j). The activation cost is low (7), whereas compensation is high (16). Agency3 commits to provide shielding and call helicopters, if the prediction is made (C_k). Both activation and compensation are very expensive.

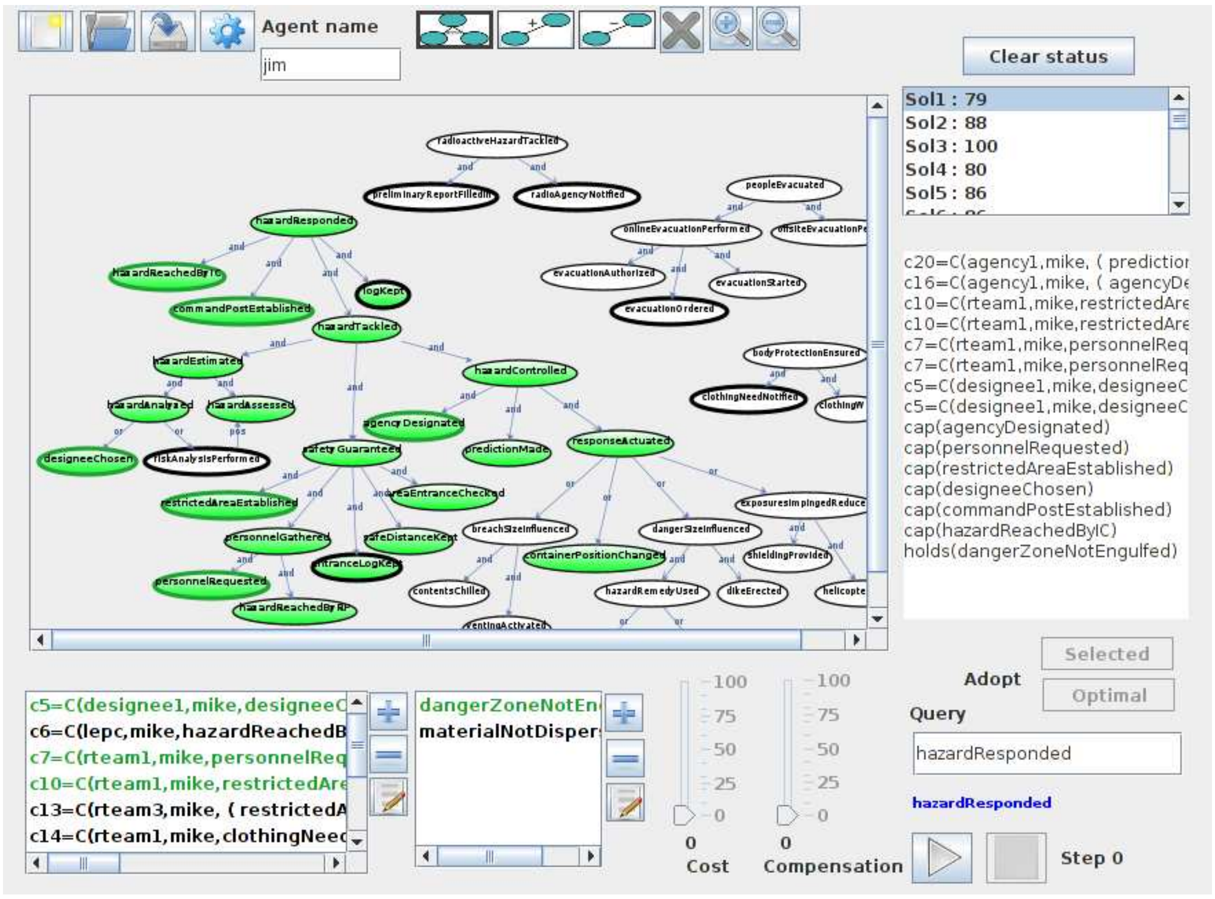
Capability	AC	CC	Capability	AC	CC
hazardReachedByIC	9	6	commandPostEstablished	8	15
designeeChosen	6	8	riskAnalysisPerformed	15	1
restrictedAreaEstablished	7	7	personnelRequested	6	9
entranceLogKept	7	0	agencyDesignated	10	14
logKept	10	0	preliminaryReportFilledIn	4	0
radioAgencyNotified	6	16	evacuationOrdered	8	14
clothingNeedNotified	6	4			

Table 7.7: Capabilities in the hazard materials response simulation

Table 7.7 lists activation and compensation costs for Mike's capabilities. Notice that the cost for a capability might vary over time, and it typically depends on available resources. We suppose here that these values don't vary during our simulations. Some capabilities do not have significant compensation cost: entranceLogKept, logKept, and preliminaryReportFilledIn. Indeed, compensating these activities means simply ignoring the data that was filled in. Some other capabilities have high compensation cost: radioAgencyNotified, commandPostEstablished, agencyDesignated, and evacuationOrdered. The reason is that compensating the effects of these actions is expensive for Mike, either in terms of work (e.g. removing a command post) or money (e.g. a penalty might be assigned if an evacuation order is cancelled).

We performed our simulations using the following adaptation policy: (i) the variant selection criteria minimizes the delta from the current configuration; (ii) the new variant should differ from the current one; (iii) threatened capabilities and commitments should be avoided; (iv) failed capabilities and violated commitments should be avoided; (v) compensation cost is considered; (vi) third-party commitments can be used; (vii) the variant generation timeout is 5 seconds, the tick time is 7 seconds; (viii) the considered adaptation triggers are all risks plus unthreatened commitments and capabilities; (iv) opportunities are adopted irrespective of their delta value.

Currently, states of the world dangerZoneNotEngulfed and materialNotDispersed hold. Mike's goal is hazardResponded. Mike is adopting a variant to support hazardResponded. We generate this variant using our prototype. The prototype outputs 39 variants; the current one (shown in Figure 7.8) has minimal cost (79). The commitment labels in the figure are numbered (C_1, C_2, ...) instead of being ordered alphabetically (C_a, C_b, ...) as in Table 7.6. Our description refers to the alphabetical version.



```
Sol1 : 79
Sol2 : 88
Sol3 : 100
Sol4 : 80
Sol5 : 86
```

```
c20=C(agency1,mike, ( predictior
c16=C(agency1,mike, ( agencyDe
c10=C(rteam1,mike,restrictedAre
c10=C(rteam1,mike,restrictedAre
c7=C(rteam1,mike,personnelReq
c7=C(rteam1,mike,personnelReq
c5=C(designee1,mike,designeeC
c5=C(designee1,mike,designeeC
cap(agencyDesignated)
cap(personnelRequested)
cap(restrictedAreaEstablished)
cap(designeeChosen)
cap(commandPostEstablished)
cap(hazardReachedByIC)
holds(dangerZoneNotEngulfed)
```

```
c5=C(designee1,mike,designeeC
c6=C(lepc,mike,hazardReachedB
c7=C(rteam1,mike,personnelReq
c10=C(rteam1,mike,restrictedAre
c13=C(rteam3,mike, ( restrictedA
c14=C(rteam1,mike,clothingNeec
```

```
dangerZoneNotEn
materialNotDisper
```

Query: hazardResponded
hazardResponded
Cost: 0
Compensation: 0

Figure 7.8: Initial variant for the simulations

To achieve his root goal, Mike exploits the world state dangerZoneNotEngulfed. He uses his capabilities, some of which are used to make other agents unconditionally committed to deliver some service. He uses his capabilities for hazardReachedByIC and for command-PostEstablished. Then, Mike uses his capability for designeeChosen to detach C_a and make Designee1 unconditionally committed to logKept and hazardAssessed. By using his capability for personnelRequested, he detaches C_c and makes Rteam1 unconditionally committed for safeDistanceKept and hazardReachedByRP. The capability for restrictedAreaEstablished is used to detach C_d and make Rteam1 unconditionally committed for entranceLogKept and areaEntranceChecked. Using his capability for agencyDesignated and supporting hazardAssessed via C_a, Mike detaches C_g and supports predictionMade. Finally, he detaches C_i and makes Agency1 unconditionally committed to containerPositionChanged.

Simulation 4 Mike is currently responding to the hazard using the variant described above. After some time, Agency1 notifies Mike that commitment C_i is at risk, for the agency is encountering difficulties in moving the HazMat container to a safe position. Mike interprets such message as a threat for C_i, and changes the state of such commitment. □

Mike's adaptation policy specifies that threatened commitments are adaptation triggers. Consequently, variant generation is executed, invoking $\mathcal{EL}^+$2SAT with a timeout of 5 seconds. The tool generates 17 possible variants. The less expensive one is the second one (cost 10), which is the following:

C_j=C(Agency2, Mike, predictionMade ∧ materialNotDispersed, dikeErected)
C_j=C(Agency2, Mike, predictionMade ∧ materialNotDispersed, fogSpraysUsed)
C_g=C(Agency1, Mike, agencyDesignated ∧ hazardAssessed, predictionMade)
C_d=C(Rteam1, Mike, restrictedAreaEstablished, entranceLogKept)
C_d=C(Rteam1, Mike, restrictedAreaEstablished, areaEntranceChecked)
C_c=C(Rteam1, Mike, personnelRequested, safeDistanceKept)
C_c=C(Rteam1, Mike, personnelRequested, hazardReachedByRP)
C_a=C(Designee1, Mike, designeeChosen, logKept)
C_a=C(Designee1, Mike, designeeChosen, hazardAssessed)
cap(agencyDesignated), cap(personnelRequested), cap(restrictedAreaEstablished),
cap(designeeChosen), cap(commandPostEstablished), cap(hazardReachedByIC)
holds(materialNotDispersed)

The cost of the variant (10) is computed by summing the compensation cost of the capabilities and commitments that are not in the new solution and the activation cost for capabilities and commitments that have to be adopted. The new variant requires to compensate C_h (cost 0) and C_i (cost 3), and to adopt commitment C_j (cost 7).

Simulation 5 Mike is still responding to the emergency. While tackling the hazard with the new variant (that includes C_j), a new event occurs. Agent Agency2 sends a message to Mike telling that it will violate C_j, for the protection dike cannot be erected safely. □

As soon as the diagnostic component of the prototype detects this failure, variant generation is performed. The prototype identifies nine variants. The number of variants is decreasing given that the adaptation policy tells to ignore both threatened and failed capabilities/commitments. The best variant is the following:

C_k = C(Agency3, Mike, predictionMade, helicoptersCalled)
C_k = C(Agency3, Mike, predictionMade, shieldingProvided)
C_g = C(Agency1, Mike, agencyDesignated ∧ hazardAssessed, predictionMade)
C_d = C(Rteam1, Mike, restrictedAreaEstablished, entranceLogKept)

$\mathsf{C}_d = \mathsf{C}$(Rteam1, Mike, restrictedAreaEstablished, areaEntranceChecked)
$\mathsf{C}_c = \mathsf{C}$(Rteam1, Mike, personnelRequested, safeDistanceKept)
$\mathsf{C}_c = \mathsf{C}$(Rteam1, Mike, personnelRequested, hazardReachedByRP)
$\mathsf{C}_a = \mathsf{C}$(Designee1, Mike, designeeChosen, logKept)
$\mathsf{C}_a = \mathsf{C}$(Designee1, Mike, designeeChosen, hazardAssessed)
cap(agencyDesignated), cap(personnelRequested), cap(restrictedAreaEstablished)
cap(designeeChosen), cap(commandPostEstablished), cap(hazardReachedByIC)

The cost for this variant is 42, and it consists of the activation cost for C_k (26) and the compensation cost for C_j (16).

Simulation 6 Mike is using the variant with C_k. Two relevant events occur during the same time tick. Commitment C_a is threatened (agent Designee1 tells it cannot keep the log), whereas Agency1 says that it is now able to bring about C_i. Notice that C_i had not been violated; however, Mike had cautiously marked it as a threatened commitment. □

The prototype starts an adaptation process triggered both by the threat (C_a) the opportunity (C_i). The variant generator outputs only one variant within the time limit:

$\mathsf{C}_i = \mathsf{C}$(Agency1, Mike, predictionMade $\wedge$ dangerZoneNotEngulfed, containerPositionChanged)
$\mathsf{C}_g = \mathsf{C}$(Agency1, Mike, agencyDesignated $\wedge$ hazardAssessed, predictionMade)
$\mathsf{C}_d = \mathsf{C}$(Rteam1, Mike, restrictedAreaEstablished, entranceLogKept)
$\mathsf{C}_d = \mathsf{C}$(Rteam1, Mike, restrictedAreaEstablished, areaEntranceChecked)
$\mathsf{C}_c = \mathsf{C}$(Rteam1, Mike, personnelRequested, safeDistanceKept)
$\mathsf{C}_c = \mathsf{C}$(Rteam1, Mike, personnelRequested, hazardReachedByRP)
cap(logKept), cap(agencyDesignated), cap(personnelRequested)
cap(restrictedAreaEstablished), cap(riskAnalysisPerformed), cap(commandPostEstablished)
cap(hazardReachedByIC), holds(dangerZoneNotEngulfed)

The delta cost of this variant—the difference from the previous variant—is 82. This derives from the adoption of commitment C_i (5), the compensation of C_a (4) and C_k (40), the exploitation of capabilities for riskAnalysisPerformed (15) and logKept (10), and the compensation cost for capability designeeChosen (8).

These simulations demonstrate that the prototype can effectively deal with a variety of events caused by the volatility of the social context. The simulations that we presented perform adaptations in response to threats for the fulfilment of commitments, violations of commitments, and opportunities caused by ceased threats for commitments. The third simulation testifies how the architecture provides self-optimization by exploiting opportunities. Adaptations consist of engaging in new commitments with other agents, releasing debtors from current commitments, and using a different subset of internal capabilities. The prototype has managed to identify some solutions within a small amount of time (the timeout was set to five seconds). Specifically, the first solution is identified after a few milliseconds. Also, we have shown how the cost-based adaptation algorithm works and preserves stability by minimizing changes between the current variant and the new one.

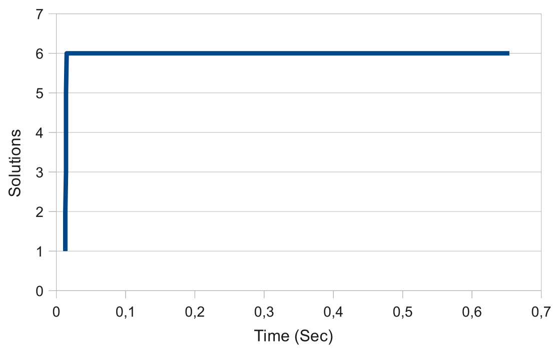
7.2.3 Scalability experiments for the second prototype

We report here on scalability experiments about the prototype for settings with social variability (Section 6.2). We verify the scalability of variant generation. We test the performance of the prototype both on variant versions of the case study and on artificial models of growing size. At the end of the section we briefly report on experiments where we exploit the DLV-complex solver on a restricted version of the language.

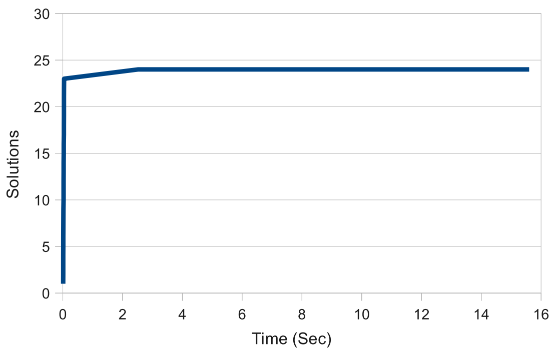
Notice that the variant generation problem is NP-complete. Since we allow for commitments expressed in propositional logic, we encode the variant generation problem as a boolean satisfiability (SAT) problem. It is well known that SAT is a NP-complete problem, which is therefore exponential in the worst case [GJ79].

The performance of our tool depends on: (i) the efficiency of $\mathcal{EL}^+$2SAT and its typical solution detection distribution; and (ii) the efficiency of our encoding of the problem. $\mathcal{EL}^+$2SAT has some distinctive features to consider: (i) if the problem is satisfiable, one solution is returned during the first SAT-solver invocation; (ii) in the typical distribution of the solutions a large amount of solutions is returned in a short time, then fewer and fewer solutions are identified. In theory, our encoding should slightly affect these properties. In particular, the first feature of $\mathcal{EL}^+$2SAT is invalid: the first solution returned by $\mathcal{EL}^+$2SAT might be discarded by the post-processing performed by Algorithm 5.5.

The basic unit to assess scalability is the number of boolean variables to which the tool assigns truth values. These variables represent states of the world, agent capabilities, and commitments. A single commitment can be mapped to more variables (Algorithm 6.2).



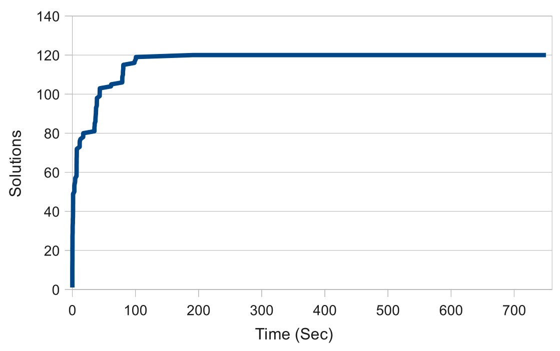
(a)



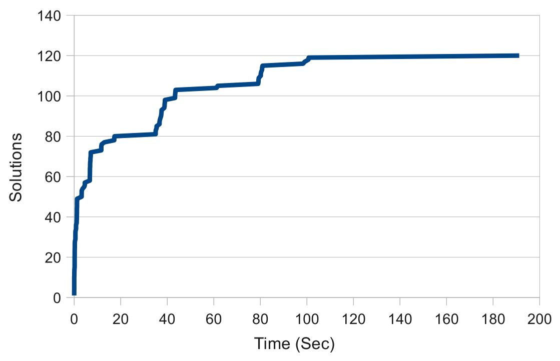
(b)

Figure 7.9: Variant generation for the case study: (a) 16 variables, 6 solutions; (b) 20 variables, 25 solutions

We perform scalability experiments on some variations of the case study. Figure 7.9 presents results for (a) a setting with 7 commitments and 7 capabilities, resulting in 16 boolean variables and 6 solutions; (b) a setting with 8 commitments and 9 capabilities, resulting in 20 variables and 25 solutions. The x axis shows time in seconds; the y axis represents solutions. Both charts show the optimal behaviour of $\mathcal{EL}^+$2SAT, where most solutions are identified at the beginning. In particular, with these small problems, most time is spent to generate new truth assignments after all solutions are found (as the line becomes horizontal). In the first example, all solutions are found in a few milliseconds, and the solver terminates after less than 0.7 seconds. In the second example, the tool takes slightly less than 16 seconds, but all solutions are identified in less than three seconds.



(a)

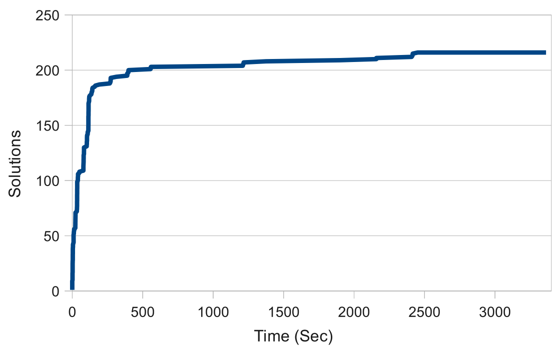


(b)

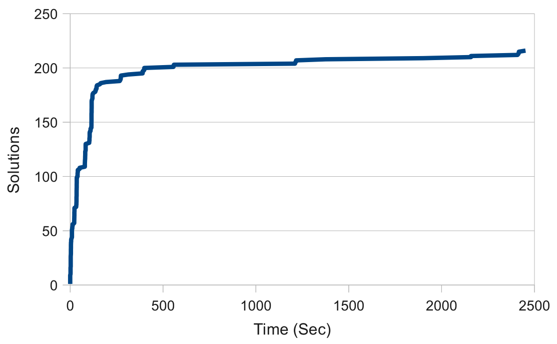
Figure 7.10: Variant generation for the case study with 25 variables and 120 solutions: (a) entire execution; (b) zoom till the last solution is found

Figure 7.10 considers a larger version of the case study with 11 commitments, 9 capabilities, and one state of the world. This setting is encoded to 25 boolean variables and has 120 solutions. Figure 7.10a shows the entire execution of the tool, whereas Figure 7.10 zooms on the time interval till the last solution. The tool takes more than 700 seconds to terminate, though all solutions are found within 200 seconds. This trend might seem negative, but should be read keeping in mind that the problem is of exponential nature. Moreover, most solutions are identified in the very beginning, then the chart line flattens. The right-hand side of the figure shows that solutions are identified in blocks: as soon as a solution is found, several small variations of the same solution are identified. This is due to optimization strategies implemented by $\mathcal{EL}^+$2SAT.

Figure 7.11 considers the setting we used in the simulations in Section 7.2.2. In particular, we analyse here the initial variant generation. In the example there are 11 commitments, 9 capabilities, and two states of the world. This setting results in 27 variables and 216 solutions. The solution determination distribution does not differ much from the previous cases. More than one hundred solutions are identified in about half a minute, then most solutions (200) are determined within 500 seconds, finally the tool identifies the remaining few solutions in a much longer time.



(a)



(b)

Figure 7.11: Variant generation for the case study with 27 variables and 216 solutions: (a) entire execution; (b) zoom till the last solution is found

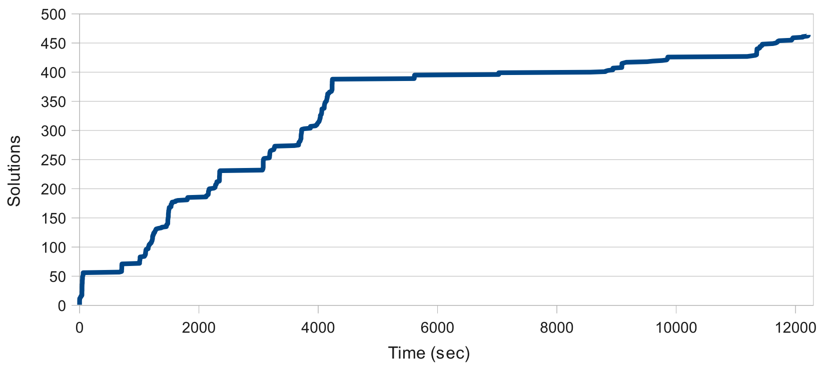
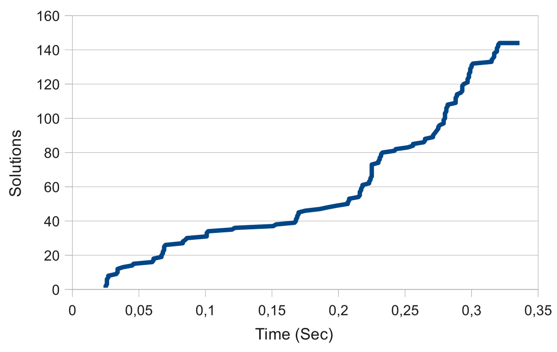


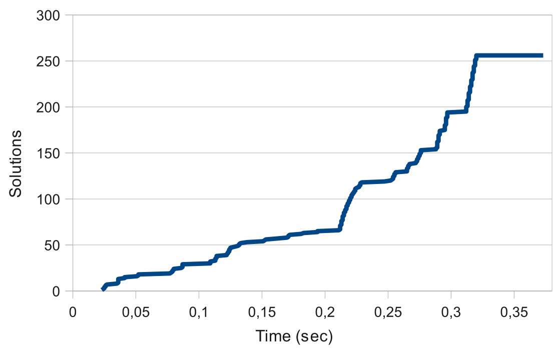
Figure 7.12: Variant generation for the case study with 32 variables, timeout at 12500 seconds

Our scalability analysis on the case study terminates with Figure 7.12: 15 commitments, 9 capabilities, and three states of the world. We ran the tool for 12500 seconds, then we manually stopped it. The tool generated 464 solutions. The shape of the chart resembles the previous ones, but the number of solutions does not grow so quickly. Given the current encoding, the tool cannot terminate in reasonable time for larger variations of the problem. However, it seems able to generate a significant number of variants.

We report now on scalability experiments where we artificially increment the number of variants. We enrich an initial goal model with an increasing number of sub-goals for which the agent has capabilities. Figure 7.13 shows variant generation for two settings where we have (a) 14 variants and 144 solutions; (b) 16 variables and 256 solutions. The trend is different from the previous settings. The reason is that a higher percentage of truth assignments leads to a solution (the density of solutions in higher). The distribution trend is quite linear: solutions are identified following an uniform pattern over time. In both cases, the tool is very efficient and terminates in less than 0.4 seconds.



(a)



(b)

Figure 7.13: Variant generation, increasing the number of solutions: (a) 14 variables and 144 solutions; (b) 16 variables and 256 solutions

Figure 7.14 presents results for larger models. We do not detail all the diagrams here. The variant identification trend resembles that of Figure 7.13, since many truth assignments are actual solutions. In this second set of experiments, we are applying the SAT-based solver to a simpler problem which could be expressed using a smaller subset of propositional logic. However, the tool is still able to identify some solutions since the beginning. This is a very good feature if one is interested in obtaining *some* solutions.

Experiment	1sec		3sec		10sec		30sec		100sec	
16-6	6	100%	6	100%	6	100%	6	100%	6	100%
20-24	23	95.8%	24	100%	24	100%	24	100%	24	100%
25-120	36	30%	49	40.8%	72	60%	80	66.7%	117	97.5%
27-216	9	4.2%	26	12%	51	23.6%	71	32.9%	130	60.2%
32-n.a.	9	n.a.	13	n.a.	13	n.a.	17	n.a.	57	n.a.
14-144	144	100%	144	100%	144	100%	144	100%	144	100%
16-256	256	100%	256	100%	256	100%	256	100%	256	100%
16-324	324	100%	324	100%	324	100%	324	100%	324	100%
18-576	50	8.7%	374	64.9%	576	100%	576	100%	576	100%
19-874	58	6.6%	74	8.5%	874	100%	874	100%	874	100%
20-1024	54	5.3%	126	12.3%	1024	100%	1024	100%	1024	100%
21-1600	23	1.4%	56	3.5%	137	8.6%	1260	78.8%	1600	100%
23-2000	23	1.2%	75	3.8%	177	8.9%	353	17.7%	2000	100%

Table 7.8: Scalability results showing the progress with different timeouts. The first part refers to increasing commitments in the case study; the second part refers to increasing capabilities

Table 7.8 summarizes scalability results. It shows the effectiveness of the tool with different timeouts. The first column is an identifier for the experiment (number of variables, number of solutions). Then, for each timeout (1, 3, 10, 30, and 100 seconds), we report

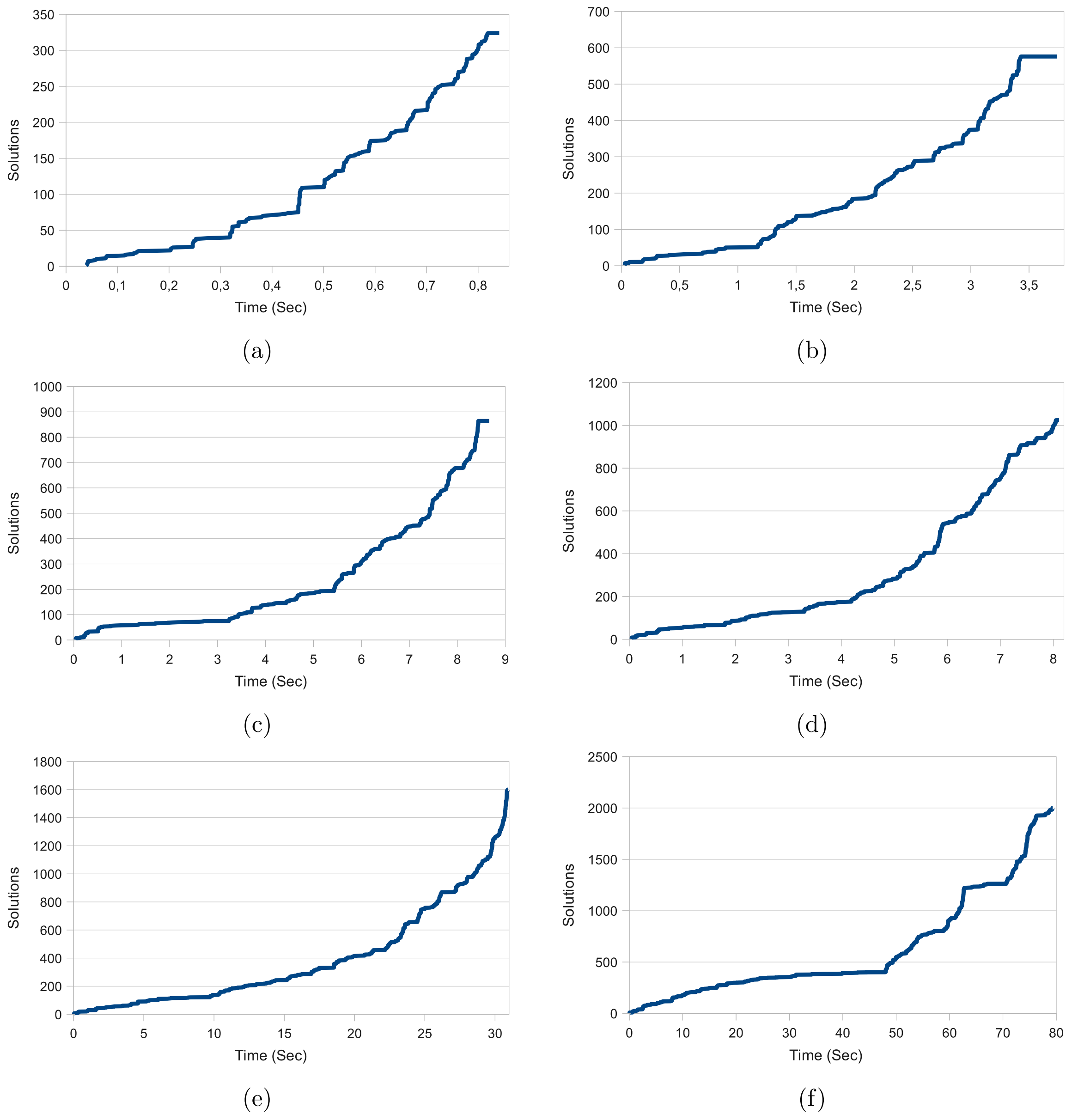


Figure 7.14: Variant generation, increasing the number of solutions: (a) 16 variables and 324 solutions; (b) 18 variables and 576 solutions; (c) 19 variables and 874 solutions; (d) 20 variables and 1024 solutions; (e) 21 variables and 1600 solutions; (f) 23 variables and 2000 solutions

the total number of generated variants and the ratio generated variants over all variants.

In all experiments, the tool always finds some solutions within one second. In the first experiment, it returns all six solutions. In other experiments, it returns a number of variants in between 9 (experiment with 32 variables) and 324 (experiment with 16 variables and 324 variants). In large experiments (e.g. 21-1600 and 23-2000), more than twenty variants are generated. With the three-seconds limit, we can notice a substantial improvement for the experiment 18-576, where more than 300 variants are generated in two seconds. All the other experiments have a linear increase in the number of variants. Within the ten-seconds time limit all solutions for experiments 19-874 and 20-1024 are computed. In the experiments based on the case study, the progress is linear. Most variant generation experiments terminate within 30 or 100 seconds.

Dealing with the exponential growth of the problem

In our scalability experiments, we applied a simple type of heuristic to deal with the exponential growth of the problem size. We specified a time limit that stops variant generation after a fixed time. Several *heuristics* can be defined to cope with the exponential nature of the problem. Another simple option is to define a minimum and maximum number of solutions. Again, variant generation can be terminated if no variants are identified for a long period of time (time might be an absolute value or based on the average time to compute the last n solutions). Devising these heuristics is part of future work.

Another way to deal with scalability issues is to limit the expressiveness of the language used to represent commitments. We have conducted some initial experiments in this direction. We limited the antecedent to a disjunction of states of the world, and the consequent to a conjunction of states. This way, we can use the disjunctive datalog solver DLV-complex to generate variants. For details about the encoding, look at [CDGM10a].

We evaluate the applicability of our reasoning to medium- and large-sized scenarios by increasing both the size of goal models and the number of commitments in the setting. We create our experiments using a scenario cloning technique: a basic building block is cloned to obtain larger scenarios. The building block consists of a goal model with 9 goals (with one top-level goal, 3 AND-decompositions and 1 OR-decomposition) and two commitments. Cloning this scenario produces a new scenario with 2 top-level goals, 18 goals and 4 commitments; another cloning operation outputs 3 top-level goals, 27 goals and 6 commitments, and so on. The query consists of the conjunction of all the top-level goals in the cloned scenario. This kind of cloning increases linearly the number of goals and commitments, whereas it increases exponentially the number of solutions.

Table 7.9 present the results of the scalability experiments. The first three columns show the number of goals, commitments and solutions, respectively. The number of solutions grows exponentially: the largest experiment has almost two millions solutions. The fourth column shows the time to generate variants; the reasoning is applicable at design time to large models, given that 2 millions solutions are computed in 200 seconds on a desktop computer. The most significant result, however, is in the last column. It shows the average time to derive one solution in microseconds. The time per solution does not grow exponentially. The average time for smaller experiments is higher because

# goals	# comms	# solutions	time (s)	$\frac{time}{\#sol}$ (μs)
9	2	5	0.009	1866
18	4	25	0.013	533
27	6	125	0.033	266
36	8	625	0.112	179
45	10	3125	0.333	107
54	12	15625	1.361	87
63	14	78125	7.017	90
72	16	390625	37.043	95
81	18	1953125	199.920	102

Table 7.9: Scalability results for variant generation with restricted syntax

initialization time has a strong impact; time grows pseudo-linearly for larger experiments.

The conclusion we draw from scalability analysis is that there is no variant generation algorithm fits perfectly to all types of settings. If the STS requires the usage of an expressive language, the SAT-based solution is the most adequate one. It returns a significant number of variants within reasonable time, and can be stopped using some heuristics. If the setting is characterized by high variability (thousands or millions of solutions), the only solution is to use a simpler language to express the problem. We have demonstrated the efficacy of this second solution through an implementation (based on disjunctive datalog) that deals with a restricted syntax for commitments.

7.3 Chapter summary

In this chapter we have evaluated our approach on two case studies. In Section 7.1 we have applied contextual goal modelling to a smart-home case study, we experimented the first implementation of our architecture via some simulated scenarios, and we performed scalability experiments concerning diagnosis and reconfiguration. In Section 7.2 we applied the modelling language for social variability to an hazardous materials response case study, we experimented the second prototype on some scenarios where commitments are violated and threatened, and we performed scalability experiments.

The contribution of the chapter is twofold. First, we have shown the adequacy of our modelling primitives to represent realistic scenarios characterized by contextual and social variability. The same models are then used by the prototype implementations of our architecture to perform self-adaptation in response to threats and failures. Second, we have performed scalability experiments to determine if the algorithms implemented in our implementations scale when considering larger settings. We have identified bottlenecks and proposed promising solutions to cope with them.

Chapter 8

Conclusions and future work

8.1 Conclusions

In this thesis, we proposed a novel approach to software self-adaptation founded on the notions of contextual and social variability. We claimed that and justified how these types of variability affect software requirements. Variability might result in either threats or opportunities. In our approach, self-adaptation is an evolutionary mechanism enacted by a software system to cope with the volatility of the environment and to ensure the fulfilment of system requirements.

Our framework applies to software that operates in socio-technical systems. As suggested by Ropohl [Rop99], we understand an STS as a goal-oriented system composed of a number of *interacting* subsystems. Subsystems can be either technical (software) or social (humans and organizations). Unlike traditional approaches in software and system engineering, we conceive interaction in terms of the social relationships it defines, the *social commitments* [Sin98] between subsystems.

The thesis lies in the vast and active landscape of approaches to self-adaptivity (look at Chapter 2 for an overview). Most work in the area focuses on technical aspects of adaptation—re-routing of architectural connectors, orchestration and composition of web services, replacement of current features with others—and largely ignores high-level concepts—system requirements and social relationships with other systems. Some approaches in Requirements Engineering propose self-adaptation based on goal models [FFvLP98, WMYM09, ST07, BP10]. To our knowledge, there is no work, apart from ours, that considers both requirements and social relationships.

The distinctive features of our approach to adaptivity are that (i) we explicitly take into account the purpose of software by keeping alive requirements models (extended goal models), and using them for diagnosis and variant selection; (ii) we represent the meaning of the social relationships—via commitments—between the considered software system and other social actors or technical systems; (iii) we propose a comprehensive account on adaptation, from its conceptualization to implementation and evaluation. We detail now the specific contributions of this thesis and present our conclusions.

Study of the impact of contextual and social factors on software adaptation

We have studied how contextual and social factors affect software requirements. In particular, we considered their impact at runtime. In this thesis, software is aware of its requirements, which are expressed via Tropos goal models [BPG$^+$04]. Goal models refine high-level objectives—that stakeholders assign to the system—by AND/OR-decomposing them into sub-goals. This decomposition process terminates with the identification of concrete plans (tasks) to achieve goals. Goal models are well suited for adaptation because they define a variability space [LLY$^+$06], alternative ways (variants) to achieve root goals. Such variability space is affected by various factors, among which:

– the physical context: the validity of a certain context is required to adopt sub-goals in OR-decompositions (or for tasks in a means-end decompositions), and to establish dependencies on other actors. Also, some sub-goals in AND-decompositions are necessary only in certain contexts.

– the social context: the variability space is enlarged when new actors come into play and commit to deliver services, or when existing actors offer new services. Conversely, the commitments the system has already made limit the variability space, provided that the system wants to respect them. Generally speaking, commitments violation is equally undesirable as requirements failure. Since commitments have contractual validity, their violation might lead to further commitments or to other penalties.

These two factors cannot be ignored by self-adaptive software. Doing that could lead, at the very least, to the selection and enactment of variants that do not work properly in the current physical context and that violate commitments the system has made. Such conclusion is particularly valid when one considers software operating in socio-technical systems, where technical and social subsystems depend one on another to fulfil their requirements.

Requirements modelling and reasoning framework for variability

We introduced and detailed two requirements modelling and reasoning frameworks that consider the impact of contextual and social factors on requirements. The modelling languages are functional to self-adaptation: at runtime, software exploits them to determine threats and to identify alternative variants to achieve goals. The contextual goal modelling framework represents contextual influence on goals by associating contexts to variation points in goal models. Since contexts are typically communicated as informal statements, we proposed context analysis as a refinement mechanism to identify a set of observable facts that evidence the validity of a context. The framework to represent social variability based on goal models and commitments relates goals and commitments through the notion of state of the world. An agent's goal is achieved when a certain state of the world is met, whereas a commitment is an agent's promise to bring about a state of the world (the consequent) if another state of the world holds (the antecedent). Such observation enables us to formally link goals and commitments and to define a formal semantics for the concept of variant. A variant for an agent's goals is a triple that consists of (i) a set of (sub)-goals the agent intends to achieve, (ii) the capabilities the agent intends to exploit to achieve those (sub)-goals; (iii) the commitments the agent intends to make to and take from other agents.

The two proposed frameworks enable software designers to represent and analyse context and social relationships (commitments), as well as to reason about their impact on variability. Understanding variability in terms of requirements—here, goals—guarantees that software self-adaptation is performed to better satisfy the current requirements.

Conceptual architecture for self-adaptive software

We devised a conceptual architecture for self-adaptive software that operates in STSs. The adaptation control loop of our architecture consists of a Monitor-Diagnose-Reconcile-Compensate (MDRC) cycle. The key features of our proposal are: (i) characterizing desired system behaviour via models that represent requirements, context, and social relations; and (ii) considering the autonomy and heterogeneity of other subsystems (both technical and social) in the STS throughout the MDRC cycle.

Our architecture supports a system in achieving its requirements. In doing so, the architecture interacts with other entities in the STS through appropriate interfaces: context sensors provide information about contextual changes, context actuators enable to effect changes, agents represent subsystems with which the supported system can establish social relationships to satisfy goals it cannot achieve by itself.

Requirements models are used at runtime to (i) diagnose failures and under-performance, and (ii) generate alternative variants that enable the system to achieve its goals. We show how the architecture can exploit the requirements models proposed in this thesis. The conceptual architecture can be instantiated using different models, e.g. other goal modelling frameworks (e.g. KAOS [DvLF93] or GBRAM [AP98]) or social interaction models (e.g. Hermes [CW06]).

Algorithms for diagnosis and reconciliation

We proposed and implemented algorithms for diagnosis and reconciliation that are part of the MDRC cycle:

– diagnosis algorithms detect plan and goal failures when plans are not carried out as expected or they do not produce the desired effect (achieving some goal). Also, they identify the violation of commitments based on timeouts and cancellation messages sent by committed agents.

– a variant selection algorithm specific for contextual variability. The selection of the best variant—among those that fit with the current context—depends on the contribution to a set of prioritized soft-goals. The current context does not only make some variants inapplicable, but also affects their quality. Variant stability is also considered: new variants that are similar to the current one are preferred over variants that are radically different.

– a variant selection algorithm specific for setting with social variability. Variants consist of capabilities and commitments to/from other agents. The selection of the best variant is based on cost. Both capabilities and commitments have (i) an activation cost, the effort needed to exploit a capability or take/make a commitment; and (ii) a compensation cost to deactivate the plan used for a capability or to cancel a commitment. The overall variant cost can be computed either summing the costs of each capability and commitment or considering only the differences from the current variant (added and removed capabilities/commitments).

The proposed algorithms provide general mechanisms for diagnosis and reconciliation. We expect that, in practice, they will be refined, customized, and optimized. For instance, the cost-based framework might be enriched with a cost computation framework.

Experimental evaluation of the approach

We evaluated our approach via case studies and scalability tests. We applied the modelling languages to represent case studies, performed simulations to demonstrate how our prototypes respond to failures and threats via self-adaptation, and conducted scalability experiments to check if our algorithms and prototypes can cope with larger problems. We repeated this evaluation methodology twice:

– we applied contextual goal modelling to a case study concerning assisted living in smart-homes. We modelled a smart-home supervisor that helps people living in the house carry out everyday activities. We performed simulations that show how our prototype reacts to plan failures and context changes. Our scalability experiments demonstrate that the diagnostic component scales in a pseudo-linear manner with respect to the number of goals and plans, and that variant generation scales linearly with the number of variants.

– we applied the framework for social variability to a case study concerning response to hazardous materials accidents. We have represented the perspective of a hypothetical incident commander that has to interact with other people to deal with the emergency. We showed, via simulations, that our prototype can successfully react to different threats, to failed capabilities, and to violated commitments. The variant generation problem with commitments, being NP-complete, is hard to tackle. Nevertheless, the scalability experiments demonstrated that the underlying automated reasoning tool—based on a state of the art SAT-solver—is able to identify a reasonable number variants in short time even with large problems. To further improve our approach, we sketched a simple heuristic and we conducted preliminary experiments with a less expressive language for commitments.

8.2 Ongoing work and future directions

This thesis opens the doors to several research lines. We introduce some directions we are working on and others we intend to investigate in the future.

Early diagnosis of failures

The adage "Prevention is better than cure" applies to software too. Early diagnosis is the detection of possible failures and threats before they occur. Self-adaptive software can anticipate the problem and actuate strategies to avoid it. This is doable if software is able to identify the symptoms of possible problems.

A way to identify these early symptoms is performing risk analysis. This technique assesses the level of risk based on positive or negative events. Risk analysis has already been applied to goal modelling, e.g. by the Goal-Risk Framework [AGM11]. This design-time framework enables to model the positive or negative impact of events on requirements (goals) and the effect of mitigation treatments on these events. Also, it supports probabilistic reasoning to quantitatively assess risks.

The Goal-Risk Framework is a valid baseline to perform early diagnosis. However, assessing risks at runtime is harder than doing it at design-time, since software operates in a volatile context. At runtime, the impact of a certain event on a goal changes over time, as well as the success of a specific treatment to mitigate a risk. Traditional risk analysis can be complemented by learning mechanisms, so that software updates its knowledge base and performs risk assessment based on up-to-date evidence. This requires software to monitor events, treatments, and their effect.

Another factor might be the reputation of other participants in the STS. For instance, if actor P is currently relying on the commitments made by a service provider S, and the reputation trend of S is negative, P might decide to adapt by switching to another service provider. Such adaptation occurs in response to early diagnosis, since S has not violated its commitment to P yet.

Represent and reason about QoS

Commitments are promises to deliver services. An important attribute of any service is its quality (Quality of Service, QoS). Different service providers commit to deliver the same service with different QoS levels. For example, agent SP_1 commits to repair your car within five days, SP_2 commits to repair it within eight days, SP_3 commits to repair it between three and six days.

QoS is a relevant factor for adaptation. If the considered agent needs a service, QoS is a discriminant to choose among different providers on the basis of those qualities the agent cares of. An agent might not be concerned with time, while it might be most interested is high-quality repairs. If the considered agent offers a service, offering a high QoS might have negative impact on other goals or other offered services. For instance, a car repairer might not have enough resources to commit to repair two cars within one day.

To support QoS in our adaptation framework, we need to (i) extend the formalization of commitment to represent QoS; (ii) provide reasoning mechanisms that consider QoS during variant generation and selection; and (iii) associate measurement instruments to qualities to detect violations in the QoS agreement.

Secure adaptation

Security is a desirable feature in any system. Self-adaptive software cannot neglect the importance of security concerns. We will investigate the concept of *secure adaptation* to identify a special class of adaptive systems where security is a primary concern.

Threats to security properties (e.g. privacy, integrity, authenticity, availability, etc.) trigger an adaptation process. During variant generation and selection, the ranking of variants takes into account how well they guarantee security properties. Secure adaptation should consider not only security qualities expressed in commitments, but also trust relations between agents. For instance, a service provider SP_1 might offer a commitment with better security than SP_2, but the service requester distrusts SP_1 due to previous unsuccessful interactions. Thus, the service requester would typically choose SP_2.

This research thread is vast and includes many research questions. First, which are the security properties relevant to self-adaptation in STSs? Second, how is trust computed and who is responsible for its computation? Third, how can an agent specify its own security policies? A policy language should be defined to express the security concerns the agent is interested in. Fourth, what should an agent do if there is no variant that satisfies security policies? Two naive options are adopting a variant that does not match its security policy and dropping/delaying current goals.

API for commitments-aware software agents

Our adaptation framework enables agents to define variants composed of capabilities and commitments. However, we did not focus on how agents interact via commitments operations (create, discharge, cancel, release, delegate, assign, etc.). A future direction is the development of an API that enables to program agent interaction. This API should offer primitives to create a commitment to another agent, to cancel previously made commitments, assign them to other agents, and so on.

The API consists of a middleware that (i) dispatches messages from the agent that performs a commitments operation to the receivers; and (ii) determines the status of commitments (conditional and unconditional ones, violated, delegated, etc.). The API should provide the publicly verifiable status of the commitment. On top of that, each agent might define additional states used for internal processing (e.g. threatened).

In Section 3.4.2 we have shown how interaction protocols are defined in terms of roles, while they are enacted when specific agents play those roles. Agent adaptation might consider the notion of role compliance: is agent x playing role ρ behaving as expected by its role? The middleware might act as the socio-legal context where agents operate: it determines violations and can impose penalties on non-compliant agents. For example it might decrement the agent's reputation, create further commitments, disable the agent.

Efficient variant generation/selection

We have demonstrated how generating and selecting variants using goal models are computationally hard tasks. Such tasks become even harder when commitments are considered. Such complexity is due to the expressiveness of the language (goal trees and propositional commitments). Devising means to cope with this issue is a main concern. There are two main strategies to improve variant generation and selection:

– *Restrict language expressiveness.* This corresponds to simplifying the problem by using a subset of the language. The language to express the antecedent and the consequent in commitments might be restricted. We performed and reported about promising experiments in Section 7.2.3. Replacing contribution to soft-goals with costs for tasks/goals is another option. This enables to explore a goal tree bottom-up in linear time; on the contrary, it does not enable to represent cross-cutting qualities.

– *Use heuristics.* Instead of determining all solutions and select the best, heuristics can be employed to identify good enough solutions. We have already discussed the use of time limits and the introduction of thresholds for the number of solutions. Another heuristic is to explore goal models top-down and make sub-optimal decisions for each encountered variation point. Again, one might define the concept of good-enough variant and stop variant generation as soon a good-enough variant is found.

Policy language

We did not propose a complete policy language for adaptation. However, this is an important future direction: policies enable fine-grained tuning of the self-adaptation mechanisms. Adaptation policies apply to all the steps of the MDRC control loop:

– Monitoring policies express what to monitor and how to do it. Depending on the currently available resources, monitoring can be fine- or coarse-grained. For example, plan monitoring might be disabled for non-critical goals. Also, less accurate but more efficient sensors might be exploited, when needed.

– Diagnosis policies tell when adaptations are triggered. Failures can be ignored if they relate to non-critical goals or occur for the first time. On the contrary, they cannot be ignored if a critical goal is involved or they occur several times. Also, priorities might be assigned to different failures. Pimentel *et al.* [PSC10] have designed and implemented a module for our architecture based on tolerance policies to specify when to ignore failures.

– Reconciliation policies determine how to search for variant behaviours. If the system has to adapt urgently, one of the first generated variants should be selected. Otherwise, the system might wait for all variants before selecting the one to enact. Reconciliation policies might tell to avoid interactions with distrusted agents or having low reputation.

– Compensation policies guide the system in switching from the current variant to the new one. These policies should define the scheduling for the activities to perform: execute plans, establish commitments, revert the effects of started or failed plans, and so on. These activities can be interleaved in different ways, time limits can be specified, precedence constraints should be considered.

Bibliography

[ACD$^+$03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services. Technical report, IBM, 2003.

[ADG08a] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Location-based Software Modeling and Analysis: Tropos-based Approach. In Qing Li, Stefano Spaccapietra, Eric Yu, and Antoni Olivé, editors, *Proceedings of the 27th International Conference on Conceptual Modeling (ER 2008)*, volume 5231 of *LNCS*, pages 169–182. Springer, 2008.

[ADG08b] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Location-based Variability for Mobile Information Systems. In Zohra Bellahsene and Michel Léonard, editors, *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE'08)*, volume 5074 of *LNCS*, pages 575–578. Springer, 2008.

[ADG09] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. A Goal Modeling Framework for Self-Contextualizable Software. In *Proceedings of the 14th International Conference on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2009)*, volume 29 of *LNBIP*, pages 326–338. Springer, 2009.

[ADG10] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. A Goal-based Framework for Contextual Requirements Modeling and Analysis. *Requirements Engineering*, 15(4):439–458, 2010.

[AdLMW09] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. Modeling Dimensions of Self-Adaptive Software Systems. In Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 27–47. Springer, 2009.

[AGM11] Yudistira Asnar, Paolo Giorgini, and John Mylopoulos. Goal-driven Risk Assessment in Requirements Engineering. *Requirements Engineering*, 2011. To appear.

[AHP94] Brent Agnew, Christine Hofmeister, and James Purtilo. Planning for Change: A Reconfiguration Language for Distributed Systems. *Distributed Systems Engineering*, 1(5):313–322, 1994.

[Ali10] Raian Ali. *Modeling and Reasoning about Contextual Requirements: Goal-based Framework*. PhD thesis, University of Trento. Information technology and telecommunications international doctoral school, http://eprints-phd.biblio.unitn.it/288/, 2010.

[Ant06] Richard J. Anthony. A Policy-Definition Language and Prototype Implementation Library for Policy-based Autonomic Systems. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC 2006)*, pages 265–276. IEEE, 2006.

[AP98] Annie I. Antón and Colin Potts. The Use of Goals to Surface Requirements for Evolving Systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 157–166. IEEE Computer Society, 1998.

[ASM04] Timo Asikainen, Timo Soininen, and Tomi Männistö. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. In Frank van der Linden, editor, *Software Product-Family Engineering*, volume 3014 of *LNCS*, pages 225–249. Springer, 2004.

[AST09] Reza Asadollahi, Mazeiar Salehie, and Ladan Tahvildari. StarMX: A Framework for Developing Self-Managing Java-based Systems. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009)*, pages 58–67, 2009.

[Aus62] John L. Austin. *How to Do Things with Words.* Clarendon Press, Oxford, 1962.

[BCP$^+$01] Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. MBP: a Model Based Planner. In *Proceedings of IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, pages 93–97, 2001.

[BDNGG07] Luciano Baresi, Elisabetta Di Nitto, Carlo Ghezzi, and Sam Guinea. A Framework for the Deployment of Adaptable Web Service Compositions. *Service Oriented Computing and Applications*, 1(1):75–91, 2007.

[Ben75] Ludwig Benner. DECIDE in Hazardous Materials Emergencies. *Fire Journal*, 69(4):13–18, 1975.

[BG06] Volha Bryl and Paolo Giorgini. Self-Configuring Socio-Technical Systems: Redesign at Runtime. *International Transactions on Systems Science and Applications*, 2(1):31–40, 2006.

[BGM06] Volha Bryl, Paolo Giorgini, and John Mylopoulos. Designing Cooperative IS: Exploring and Evaluating Alternatives. In *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS'06)*, volume 4275 of *LNCS*, pages 533–550. Springer, 2006.

[BGM09] Volha Bryl, Paolo Giorgini, and John Mylopoulos. Designing Socio-Technical Systems: from Stakeholder Goals to SocialNetworks. *Requirements Engineering*, 14(1):47–70, 2009.

[BMSG$^+$09] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering Self-Adaptive Systems through Feedback Loops. In Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 48–70. Springer, 2009.

[BMZ$^+$05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a Taxonomy of Software Change: Research Articles. *Journal of Software Maintenance and Evolution*, 17(5):309–332, 2005.

[BP10] Luciano Baresi and Liliana Pasquale. Live Goals for Adaptive Service Compositions. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010)*, pages 114–123, 2010.

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

[BPR99] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE–A FIPA-compliant Agent Framework. In *Proceedings of the 4th International Conference on the Practical Application of Intelligent Agents and Multi Agent Technology (PAAM'99)*, pages 97–108, 1999.

[BPS10] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy Goals for Requirements-driven Adaptation. In *Proceedings of the 18th International IEEE Requirements Engineering Conference (RE 2010)*, 2010.

[Bra87] Michael E. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press Cambridge, Massachusetts, 1987.

[Bre99a] Patrick Brezillon. Context in Artificial Intelligence: I. A Survey of the Literature. *Computers and artificial intelligence*, 18(4):321–340, 1999.

[Bre99b] Patrick Brezillon. Context in Problem Solving: a Survey. *The Knowledge Engineering Review*, 14(1):47–80, 1999.

[BSP$^+$02] Joseph P. Bigus, Donald A. Schlosnagle, Jeff R. Pilgrim, Nathaniel Mills, and Yixin Diao. ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Systems Journal*, 41(3):350–371, 2002.

[BWH07] Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[BWS$^+$10] Nelly Bencomo, Jon Whittle, Peter Sawyer, Anthony Finkelstein, and Emmanuel Letier. Requirements Reflection: Requirements as Runtime Entities. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, pages 199–202. ACM, 2010.

[C$^+$99] James Clark et al. XSL Transformations (XSLT) Version 1.0. *W3C Recommendation*, 16(11), 1999.

[CCG$^+$06] Stefano Campadello, Luca Compagna, Daniel Gidoin, Silke Holtmanns, Valentino Meduri, Jean-Christophe R. Pazzaglia, Magali Seguran, and R. Thomas. Serenity Deliverable A7.D1.1: Scenario Selection and Definition, 2006.

[CCIL08] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable Functions in ASP: Theory and Implementation. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 407–424. Springer, 2008.

[CDGM10a] Amit K. Chopra, Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Modeling and Reasoning about Service-Oriented Applications via Goals and Commitments. In *Proceedings of 22nd International Conference on Advanced Information Systems Engineering (CAiSE'10)*, volume 6051 of *LNCS*, pages 113–128. Springer, 2010.

[CDGM10b] Amit K. Chopra, Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Reasoning about Agents and Protocols via Goals and Commitments. In Wiebe van der Hoek, Gal A. Kaminka, Yves Lespérance, Michael Luck, and Sandip Sen, editors, *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, pages 457–464. IFAAMAS, 2010.

[CDNFP10] Luca Cavallaro, Elisabetta Di Nitto, Carlo A. Furia, and Matteo Pradella. A Tile-based Approach for Self-assembling Service Compositions. In *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'10)*, pages 43–52. IEEE, 2010.

[CFNF97] Don Cohen, Martin S. Feather, K. Narayanaswamy, and Stephen S. Fickas. Automatic Monitoring of Software Requirements. In *Proceedings of the 19th International Conference on Software Engineering (ICSE 1997)*, pages 602–603. ACM New York, NY, USA, 1997.

[CGFP09] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. *IEEE Computer*, 42(10):37–43, 2009.

[CGS06] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based Self-Adaptation in the Presence of Multiple Objectives. In *Proceedings of the 2006 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2006)*, pages 2–8, 2006.

[CGS09] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. RAIDE for Engineering Architecture-based Self-Adaptive Systems. In *Companion of the 31st International Conference on Software Engineering (ICSE Companion '09)*, pages 435 –436, 2009.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[CNYM00] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-functional Requirements in Software Engineering.* Springer, 2000.

[CPGS09] Shang-Wen Cheng, Vahe Poladian, David Garlan, and Bradley Schmerl. Improving Architecture-Based Self-Adaptation through Resource Prediction. In Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 71–88. Springer, 2009.

[CS09] Amit K. Chopra and Munindar P. Singh. Multiagent Commitment Alignment. In Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Simo Sichman, editors, *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 937–944. IFAAMAS, 2009.

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

[CSWW04] David M. Chess, Alla Segal, Ian Whalley, and Steve R. White. Unity: Experiences with a Prototype Autonomic Computing System. In *Proceedings of the 1st IEEE International Conference on Autonomic Computing (ICAC 2004)*, pages 140–147. IEEE Computer Society, 2004.

[CW06] Christopher Cheong and Michael Winikoff. Hermes: Designing Goal-Oriented Agent Interactions. In Jörg Müller and Franco Zambonelli, editors, *Agent-Oriented Software Engineering VI*, volume 3950 of *LNCS*, pages 16–27. Springer, 2006.

[Dam07] Christian W. Damus. Implementing Model Integrity in EMF with MDT OCL. Eclipse Corner Articles, online at: http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html, 2007.

[DCGM10] Fabiano Dalpiaz, Amit K. Chopra, Paolo Giorgini, and John Mylopoulos. Adaptation in Open Systems: Giving Interaction its Rightful Place. In *Proceedings of the 29th International Conference on Conceptual Modeling (ER 2010)*, volume 6412 of *LNCS*, pages 31–45. Springer, 2010.

[DCS10] Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. Amoeba: A Methodology for Modeling and Evolution of Cross-Organizational Business Processes. *ACM Transactions on Software Engineering and Methodology*, 19(2), 2010.

[DD10] Christoph Dorn and Schahram Dustdar. Interaction-Driven Self-adaptation of Service Ensembles. In Barbara Pernici, editor, *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE'10)*, volume 6051 of *LNCS*, pages 393–408. Springer, 2010.

[DDL$^+$10] Schahram Dustdar, Christoph Dorn, Fei Li, Luciano Baresi, Giacomo Cabri, Cesare Pautasso, and Franco Zambonelli. A Roadmap Towards Sustainable Self-Aware Service Systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010)*, pages 10–19, 2010.

[DDVMW02] Theo D'Hondt, Kris De Voider, Kim Mens, and Roel Wuyts. Co-evolution of Object-Oriented Software Design and Implementation. In Mehmed Aksit, editor, *Software Architectures and Component Technology.* Kluwer Academic Publishers, 2002.

[Dey01] Anind K. Dey. Understanding and Using Context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

[DGM09a] Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. An Architecture for Requirements-Driven Self-Reconfiguration. In *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE'09)*, volume 5565 of *LNCS*, pages 246–260. Springer, 2009.

[DGM09b] Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Software Self-Reconfiguration: a BDI-based Approach. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 1159–1160. IFAAMAS, 2009.

[DK76] Frank DeRemer and Hans H. Kron. Programming-in-the-Large versus Programming-in-the Small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.

[DMCS05] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction Protocols as Design Abstractions for Business Processes. *IEEE Transactions on Software Engineering*, 31(12):1015–1027, December 2005.

[DMSGK06] Giovanna Di Marzo Serugendo, Marie-Pierre Gleizes, and Anthony Karageorgos. Self-Organization in Multi-Agent Systems. *The Knowledge Engineering Review*, 20(02):165–189, 2006.

[DMT09] DMTF. CIM Simplified Policy Language (CIM-SPL). Technical report, DMTF, 2009.

[DNDM09] Elisabetta Di Nitto, Daniel J. Dubois, and Raffaela Mirandola. On Exploiting Decentralized Bio-inspired Self-Organization Algorithms to Develop Real Systems. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009)*, pages 68 –75, 2009.

[DNGM$^+$08] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A Journey to Highly Dynamic, Self-Adaptive Service-based Applications. *Automated Software Engineering*, 15(3-4):313–341, 2008.

[DPTS06] Giovanni Denaro, Mauro Pezzé, Davide Tosi, and Daniela Schilling. Towards Self-adaptive Service-oriented Architectures. In *Proceedings of the 2006 Workshop on Testing, Analysis, and Verification of Web Services and Applications (TAV-WEB '06)*, pages 10–16, 2006.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed Requirements Acquisition. *Science of computer programming*, 20(1-2):3–50, 1993.

[EHN94] Kutluhan Erol, James A. Hendler, and Dana S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning. In *Proceedings of the 2nd International Conference on AI Planning Systems (AIPS 94)*, pages 249–254, 1994.

[Eld95] Niles Eldredge. *Reinventing Darwin: the Great Evolutionary Debate.* Weidenfeld and Nicolson, 1995.

[Eme59] Fred E. Emery. Characteristics of Socio-Technical Systems. Technical Report 527, London: Tavistock Institute, 1959.

[ES04] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[FCSM05] Scott D. Fleming, Betty H. C. Cheng, R. E. Kurt Stirewalt, and Philip K. McKinley. An Approach to Implementing Dynamic Adaptation in C++. In *Proceedings of the 2005 ICSE Workshop on Design and Evolution of Autonomic Application Software (DEAS 2005)*, pages 1–7, 2005.

[FF95] Stephen Fickas and Martin S. Feather. Requirements Monitoring in Dynamic Environments. In *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering (RE'95)*, pages 140–147. IEEE Computer Society Washington, DC, USA, 1995.

[FFMM94] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an Agent Communication Language. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463. ACM, 1994.

[FFvLP98] Martin S. Feather, Stephen Fickas, Axel van Lamsweerde, and Cristophe Ponsard. Reconciling System Requirements and Runtime Behavior. In *Proceedings of the 9th International Workshop on Software Specification and Design (IWSSD'98)*, pages 50–59. IEEE Computer Society Washington, DC, USA, 1998.

[FFZ05] Wang Fei and Li Fan-Zhang. The Design of an Autonomic Computing Model and the Algorithm for Decision-making. In *In Proceedings of the 2005 IEEE International Conference on Granular Computing (GrC 2005)*, pages 270–273, 2005.

[FHS+06] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, 2006.

[Fin08] Anthony Finkelstein. Requirements Reflection. Talk given at the 2008 Dagstuhl seminar on Software Engineering for Self-Adaptive Systems. Available at http://www.cs.ucl.ac.uk/staff/A.Finkelstein/talks/reqtsreflection.pdf, 2008.

[FIP97] FIPA. FIPA Specification part 2: Agent Communication Language. Technical report, Technical report, FIPA-Foundation for Intelligent Physical Agents, 1997.

[FL03] Maria Fox and Derek Long. PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20(1):61–124, 2003.

[FS01] Anthony Finkelstein and Andrea Savigni. A Framework for Requirements Engineering for Context-Aware Services. In *Proceedings of 1st International Workshop From Software Requirements to Architectures (STRAW 01)*, 2001.

[Fut05] Douglas J. Futuyma. *Evolution.* Sinauer Associates, 2005.

[Gat98] Erann Gat. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, chapter Three-layer Architectures, pages 195–210. MIT Press, Cambridge, MA, USA, 1998.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, Oct. 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Giu93] Fausto Giunchiglia. Contextual Reasoning. *Epistemologia, Special Issue on I Linguaggi e le Macchine*, 16:345–364, 1993.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, January 1979.

[GS02] David Garlan and Bradley Schmerl. Model-based Adaptation for Self-Healing Systems. In *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS 2002)*, pages 27–32, 2002.

[GW04] Hassan Gomaa and Diana L. Webber. Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. In *Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS 2004)*, 2004.

[Har03] Richard Harper. *Inside the Smart Home.* Springer, 2003.

[HdJ82] Carl E. Hewitt and Peter de Jong. Open Systems. In *In Proceedings of the 1982 Intervale Workshop*, pages 147–164, 1982.

[Hew86] Carl E. Hewitt. Offices are Open Systems. *ACM Transactions on Information Systems*, 4(3):271–287, 1986.

[Hor01] Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. Talk given at IBM TJ Watson Labs, NY, October 2001.

[HP03] Günter Halmans and Klaus Pohl. Communicating the Variability of a Software-product Family to Customers. *Software and Systems Modeling*, 2(1):15–36, 2003.

[Hub99] Marcus J. Huber. JAM: A BDI-theoretic Mobile Agent Architecture. In *Proceedings of the 3rd Annual Conference on Autonomous Agents (AGENTS'99)*, pages 236–243. ACM, 1999.

[IPT09] Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Towards an Assume-Guarantee Theory for Adaptable Systems. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009)*, pages 106–115, 2009.

[Jac00] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman, 2000.

[JMF08] Ivan J. Jureta, John Mylopoulos, and Stephane Faulkner. Revisiting the Core Ontology and Problem in Requirements Engineering. In *Proceedings of the 16th IEEE International Conference on Requirements Engineering (RE 2008)*, pages 71–80, 2008.

[JRL99] Ravi Jain, Maria C. Rivera, and James A. Lake. Horizontal Gene Transfer among Genomes: the Complexity Hypothesis. *Proceedings of the National Academy of Sciences of the United States of America*, 96(7):3801–3806, 1999.

[KA09] Philip Kotler and Gary Armstrong. *Principles of Marketing*. Prentice Hall, 2009.

[KBR$^+$04] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web Services Choreography Description Language Version 1.0. Technical report, W3C, 2004.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[KCH$^+$90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990.

[KHC02] Sanjeev Kumar, Marcus J. Huber, and Philip R. Cohen. Representing and Executing Protocols as Joint Actions. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 543–550, 2002.

[KKL$^+$98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, volume 1241 of *LNCS*, pages 220–242, 1997.

[KLS$^+$01] Gabor Karsai, Akos Ledeczi, Janos Sztipanovits, Gabor Peceli, Gyula Simon, and Tamas Kovacshazy. An Approach to Self-adaptive Software Based on Supervisory Control. In *Proceedings of the 2nd International Workshop on Self-Adaptive Software: Applications (IWSAS 2001)*, volume 2614 of *LNCS*, pages 77–92. Springer, 2001.

[KM07] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 259–268. IEEE Computer Society, 2007.

[KSSA09] Michiel Koning, Chang-ai Sun, Marco Sinnema, and Paris Avgeriou. VxBPEL: Supporting Variability for Web services in BPEL. *Information and Software Technology*, 51(2):258–269, 2009.

[KW04] Jeffrey O. Kephart and William E. Walsh. An Artificial Intelligence Perspective on Autonomic Computing Policies. In *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, pages 3–12. IEEE Computer Society, 2004.

[LAP06] Alexander Lazovik, Marco Aiello, and Mike Papazoglou. Planning and Monitoring the Execution of Web Service Requests. *International Journal on Digital Libraries*, 6(3):235–246, 2006.

[LFP99] Yannis Labrou, Tim Finin, and Yun Peng. Agent Communication Languages: The Current Landscape. *IEEE Intelligent Systems*, 14(2):45–52, 1999.

[LJL$^+$07] Sotirios Liaskos, Lei Jiang, Alexei Lapouchnian, Yiqiao Wang, Yijun Yu, Julio Cesar Sampaio do Prado Leite, and John Mylopoulos. Exploring the Dimensions of Variability: a Requirements Engineering Perspective. In *Proceedings of the 1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2007)*, 2007.

[LLY$^+$06] Sotirios Liaskos, Alexei Lapouchnian, Yijun Yu, Eric Yu, and John Mylopoulos. On Goal-based Variability Acquisition and Analysis. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE 2006)*, pages 76–85. IEEE Computer Society, 2006.

[LPH04] Hua Liu, Manish Parashar, and Salim Hariri. A Component-based Programming Model for Autonomic Applications. In *Proceedings of the 1st IEEE International Conference on Autonomic Computing (ICAC 2004)*, pages 10–17, 2004.

[LSQC05] Ying Li, Kewei Sun, Jie Qiu, and Ying Chen. Self-Reconfiguration of Service-based Systems: a Case Study for Service Level Agreements and Resource Optimization. In *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005)*, pages 266–273, 2005.

[LYLM06] Alexei Lapouchnian, Yijun Yu, Sotirios Liaskos, and John Mylopoulos. Requirements-driven Design of Autonomic Application Software. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative research (CASCON'06)*. ACM Press New York, NY, USA, 2006.

[May82] Ernst Mayr. *The Growth of Biological Thought*. Belknap Press Cambridge, MA, 1982.

[MBJK90] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing Knowledge about Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362, 1990.

[MBZR03] Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. Towards a Taxonomy of Software Evolution. In *Proceedings of the 1st Workshop on Unanticipated Software Evolution*, 2003.

[McC93] John McCarthy. Notes on Formalizing Context. In *Proceedings of the 13th International Joint Conference on Artifical intelligence (IJCAI-93)*, pages 555–560, 1993.

[MCN92] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and Using Nonfunctional Requirements: a Process-oriented Approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In Wilhelm Schfer and Pere Botella, editors, *Proceedings of the 5th European Software Engineering Conference (ESEC '95)*, volume 989 of *LNCS*, pages 137–153. Springer, 1995.

[MG05] Arun Mukhija and Martin Glinz. Runtime Adaptation of Applications Through Dynamic Recomposition of Components. In Michael Beigl and Paul Lukowicz, editors, *Proceedings of the 18th*

International Conference on Architecture of Computing Systems (ARCS 2005), volume 3432 of *LNCS*, pages 124–138. Springer, 2005.

[MKM06] Andrew McVeigh, Jeff Kramer, and Jeff Magee. Using Resemblance to Support Component Reuse and Evolution. In *Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems (SAVCBS '06)*, pages 49–56, 2006.

[Mow94] Abbe Mowshowitz. Virtual Organization: A Vision of Management in the Information Age. *The Information Society*, 10(4):267–288, 1994.

[MPP08] Mirko Morandini, Loris Penserini, and Anna Perini. Towards Goal-oriented Development of Self-Adaptive Systems. In *Proceedings of the 2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008)*, pages 9–16, 2008.

[MPS08] Hausi Müller, Mauro Pezzè, and Mary Shaw. Visibility of Control in Adaptive Systems. In *Proceedings of the 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS'08)*, pages 23–26, 2008.

[MSKC04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing Adaptive Software. *IEEE Computer*, 37(7):56–64, 2004.

[Mye99] Karen L. Myers. CPEF: A Continuous Planning and Execution Framework. *AI Magazine*, 20(4):63–69, 1999.

[MZ04] Marco Mamei and Franco Zambonelli. Self-Organization in Multi Agent Systems: A Middleware Approach. In Giovanna Di Marzo Serugendo, Anthony Karageorgos, Omer F. Rana, and Franco Zambonelli, editors, *Engineering Self-Organising Systems*, volume 2977 of *LNCS*, pages 233–248. Springer, 2004.

[Obj08] Object Management Group. Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0. Technical report, 2008.

[OGT$^+$99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 177–186, 1998.

[OMT08] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*, pages 899–910, 2008.

[OPB01] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing Agent Interaction Protocols in UML. In Paolo Ciancarini and Michael Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*, pages 201–218. Springer, 2001.

[Par97] H. Van Dike Parunak. "Go to the Ant": Engineering Principles from Natural Multi-Agent Systems. *Annals of Operations Research*, 75:69–101, 1997.

[PBD$^+$05] Relu Patrascu, Craig Boutilier, Rajarshi Das, Jeffrey O. Kephart, Gerald Tesauro, and William E. Walsh. New Approaches to Optimization and Utility Elicitation in Autonomic Computing. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 140–145, 2005.

[PBL05] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. *Multi-Agent Programming*, chapter Jadex: A BDI Reasoning Engine, pages 149–174. Springer, 2005.

[Pel03] Chris Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10):46–52, 2003.

[PG03] Mike P. Papazoglou and Dimitrios Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10):25–28, 2003.

[PLL$^+$03] Manish Parashar, Hua Liu, Zhen Li, Vincent Matossian, Cristina Schmidt, Guangsen Zhang, and Salim Hariri. AutoMate: Enabling Autonomic Applications on the Grid. In *Proceedings of the 5th Annual Workshop on Active Middleware Services (AMS 2003)*, pages 48 – 57, jun. 2003.

[PPSM07] Loris Penserini, Anna Perini, Angelo Susi, and John Mylopoulos. High Variability Design for Software Agents: Extending Tropos. *ACM Transactions on Autonomous and Adaptive Systems*, 2(4):16, 2007.

[PSC10] Joao Pimentel, Emanuel Santos, and Jaelson Castro. Conditions for Ignoring Failures Based on a Requirements Model. In *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE 2010)*, pages 48–53, 2010.

[Rao96] Anand S. Rao. AgentSpeak (L): BDI Agents Speak Out in a Logical Computable Language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*. Springer, 1996.

[RG92] Anand S. Rao and Michael P. Georgeff. An Abstract Architecture for Rational Agents. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 439–449, 1992.

[RG95] Anand S. Rao and Michael P. Georgeff. BDI Agents: From Theory to Practice. In *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319. San Fransisco, USA, 1995.

[Rob06] William N. Robinson. A Requirements Monitoring Framework for Enterprise Systems. *Requirements Engineering*, 11(1):17–41, 2006.

[Rob08] William N. Robinson. Extended OCL for Goal Monitoring. *Electronic Communications of the EASST*, 9:1–12, 2008.

[Rop99] Günter Ropohl. Philosophy of Socio-Technical Systems. *Society for Philosophy and Technology*, 4(3), 1999.

[RP09] William N. Robinson and Sandeep Purao. Specifying and Monitoring Interactions and Commitments in Open Business Processes. *IEEE Software*, 26(2):72–79, 2009.

[RSA98] Colette Rolland, Carine Souveyet, and Camille Ben Achour. Guiding Goal Modeling using Scenarios. *IEEE Transactions on Software Engineering*, 24(12):1055–1071, 1998.

[Sac77] Earl D. Sacerdoti. A Structure for Plans and Behavior. Technical report, Artificial Intelligence Center, SRI International, 1977.

[Sat01] Mahadev Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17, Aug 2001.

[SBNH05] Johanneke Siljee, Ivor Bosloper, Jos Nijhuis, and Dieter Hammer. DySOA: Making Service Systems Self-adaptive. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC 2005)*, volume 3826 of *LNCS*, pages 255–268. Springer, 2005.

[SCD09] Munindar P. Singh, Amit K. Chopra, and Nirmit Desai. Commitment-Based Service-Oriented Architecture. *IEEE Computer*, 42:72–79, 2009.

[SCDM04] Munindar P. Singh, Amit K. Chopra, Nirmit V. Desai, and Ashok U. Mallya. Protocols for Processes: Programming in the Large for Open Systems (extended abstract). In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04)*, pages 120–123, 2004.

[Sea70] John R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1970.

[Sea75] John R. Searle. Indirect Speech Acts. *Syntax and Semantics*, 3:59–82, 1975.

[SH05] Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons Inc, 2005.

[SHMK08] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From Goals to Components: a Combined Approach to Self-Management. In *Proceedings of the 2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008)*, pages 1–8, 2008.

[Sin91] Munindar P. Singh. Social and Psychological Commitments in Multiagent Systems. In *AAAI Fall Symposium on Knowledge and Action at Social and Organizational Levels*, pages 104–106, 1991.

[Sin98] Munindar P. Singh. Agent Communication Languages: Rethinking the Principles. *IEEE Computer*, 31(12):40–47, December 1998.

[Sin99] Munindar P. Singh. An Ontology for Commitments in Multiagent Systems: Toward a Unification of Normative Concepts. *Artificial Intelligence and Law*, 7(1):97–113, 1999.

[SLAT09] Mazeiar Salehie, Sen Li, Reza Asadollahi, and Ladan Tahvildari. Change Support in Adaptive Software: A Case Study for Fine-grained Adaptation. In *Proceedings of the 6th International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EaSE 2009)*, pages 35–44. IEEE Computer Society, 2009.

[SLRM10] Vitor E. Silva Souza, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. Awareness Requirements for Adaptive Systems. Technical Report DISI-10-049, DISI, University of Trento, 2010.

[SMCS04] S. Masoud Sadjadi, Philip K. McKinley, Betty H.C. Cheng, and R.E Kurt Stirewalt. TRAP/J: Transparent Generation of Adaptable Java Programs. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, pages 1243–1261. Springer, 2004.

[SP07] Sebastian Sardina and Lin Padgham. Goals in the Context of BDI Plan Failure and Planning. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007)*, pages 16–23, 2007.

[SS97] Tony Savor and Rudolph E. Seviora. An Approach to Automatic Detection of Software Failures in Real-time Systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTSA 1997)*, pages 136–146, 1997.

[ST07] Mazeiar Salehie and Ladan Tahvildari. A Weighted Voting Mechanism for Action Selection Problem in Self-Adaptive Software. In *Proceedings of the 1st IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, pages 328–331. IEEE Computer Society, 2007.

[SV09] Roberto Sebastiani and Michele Vescovi. Axiom Pinpointing in Lightweight Description Logics via Horn-SAT Encoding and Conflict Analysis. In Renate Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction (CADE-22)*, volume 5663 of *LNCS*, pages 84–99. Springer, 2009.

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A Taxonomy of Variability Realization Techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[TAHM04] Jeremy N. Timmis, Michael A. Ayliffe, Chun Y. Huang, and William Martin. Endosymbiotic Gene Transfer: Organelle Genomes Forge Eukaryotic Chromosomes. *Nature Reviews Genetics*, 5(2):123–135, 2004.

[TCW$^+$04] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A Multi-Agent Systems Approach to Autonomic Computing. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 464–471, 2004.

[Tes07] Gerald Tesauro. Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies. *IEEE Internet Computing*, 11(1):22–30, 2007.

[TP04] Paolo Traverso and Marco Pistore. Automated Composition of Semantic Web Services into Executable Processes. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*, volume 3298 of *LNCS*, pages 380–394. Springer, 2004.

[TS09] Pankaj R. Telang and Munindar P. Singh. Enhancing Tropos with Commitments. In Alex Borgida, Vinay Chaudhri, Paolo Giorgini, and Eric Yu, editors, *Conceptual Modeling: Foundations and Applications*. Springer, 2009.

[TWPF02] John Thangarajah, Michael Winikoff, Lin Padgham, and Klaus Fischer. Avoiding Resource Conflicts in Intelligent Agents. In *Proceedings of the 15th European Conference on Artifical Intelligence (ECAI 2002)*, pages 18–22, 2002.

[UBR04a] Amy Unruh, James Bailey, and Kotagiri Ramamohanarao. A Framework for Goal-Based Semantic Compensation in Agent Systems. In *Proceedings of the Workshop on Safety and Security in Multi-Agent Systems (SASEMAS '04)*, volume 4324 of *LNCS*, pages 130–146, 2004.

[UBR04b] Amy Unruh, James Bailey, and Kotagiri Ramamohanarao. Managing Semantic Compensation in a Multi-Agent System. In *Proceedings of the 12th International Conference on Cooperative Information Systems (CoopIS 2004)*, volume 3290 of *LNCS*, pages 245–263. Springer, 2004.

[VB73] Ludwig Von Bertalanffy. *General System Theory: Foundations, Development, Applications.* George Braziller New York, 1973.

[vdH04] André van der Hoek. Design-time Product Line Architectures for Any-Time Variability. *Science of computer programming*, 53(3):285–304, 2004.

[vdKdW05] Roman van der Krogt and Mathijs de Weerdt. Plan Repair as an Extension of Planning. In *Proceedings of the 2005 International Conference on Automated Planning & Scheduling (ICAPS 2005)*, pages 161–170, 2005.

[vdKdWW03] Roman van der Krogt, Mathijs de Weerdt, and Cees Witteveen. A Resource Based Framework for Planning and Replanning. *Web Intelligence and Agent Systems*, 1(3):173–186, 2003.

[vdML02] Thomas von der Maßen and Horst Lichter. Modeling Variability by UML Use Case Diagrams. In *Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL02)*, 2002.

[VG10] Thomas Vogel and Holger Giese. Adaptation and Abstract Runtime Models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010)*, pages 39–48, 2010.

[vL00] Alex van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 5–19, 2000.

[vL01] Alex van Lamsweerde. Goal-oriented Requirements Engineering: A Guided Tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE 2001)*, pages 249–263, 2001.

[vOvdLKM02] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2002.

[Wei91] Mark Weiser. The Computer for the Twenty-First Century. *Scientific American*, 265(3):94–104, 1991.

[WM09] Yiqiao Wang and John Mylopoulos. Self-repair Through Reconfiguration: A Requirements Engineering Approach. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, pages 257–268. IEEE Computer Society, 2009.

[WMYM07] Yiqiao Wang, Sheila McIlraith, Yijun Yu, and John Mylopoulos. An Automated Approach to Monitoring and Diagnosing Requirements. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 293–302. ACM New York, NY, USA, 2007.

[WMYM09] Yiqiao Wang, Sheila McIlraith, Yijun Yu, and John Mylopoulos. Monitoring and Diagnosing Software Requirements. *Automated Software Engineering*, 16(1):3–35, 2009.

[WP99] Michael Wooldridge and Simon Parsons. Intention Reconsideration Reconsidered. In *Proceeedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL'98)*, volume 1555 of *LNCS*, pages 63–79. Springer, 1999.

[YS03] Håkan L. S. Younes and Reid G. Simmons. VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research*, 20(1):405–430, 2003.

[Yu96] Eric Siu-Kwong Yu. *Modelling Strategic Relationships for Process Reengineering.* PhD thesis, University of Toronto, Toronto, Ont., Canada, Canada, 1996.

[Zad65] Lofti A. Zadeh. Fuzzy Sets. *Information and control*, 8(3):338–353, 1965.

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based Development of Dynamically Adaptive Software. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 371–380, 2006.

[ZJ97] Pamela Zave and Michael Jackson. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.