# Exploiting Requirements Variability for Software Customization and Adaptation

by

Alexei Lapouchnian

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

Department of Computer Science
University of Toronto

Exploiting Requirements Variability for
Software Customization and Adaptation

Alexei Lapouchnian

Doctor of Philosophy

Department of Computer Science
University of Toronto

2011

## Abstract

The complexity of software systems is exploding, along with their use and application in new domains. Managing this complexity has become a focal point for research in Software Engineering. One direction for research in this area is developing techniques for designing adaptive software systems that self-optimize, self-repair, self-configure and self-protect, thereby reducing maintenance costs, while improving quality of service.

This thesis presents a requirements-driven approach for developing adaptive and customizable systems. Requirements goal models are used as a basis for capturing problem variability, leading to software designs that support a space of possible behaviours – all delivering the same functionality. This space can be exploited at system deployment time to customize the system on the basis of user preferences. It can also be used at runtime to support system adaptation if the current behaviour of the running system is deemed to be unsatisfactory.

The contributions of the thesis include a framework for systematically generating designs from high-variability goal models. Three complementary design views are generated: configurational view (feature model), behavioural view (statecharts) and an architectural view (parameterized architecture). The framework is also applied to the field of business process management for intuitive high-level process customization.

In addition, the thesis proposes a modeling framework for capturing domain variability through contexts and applies it to goal models. A single goal model is used to capture requirements variations in different contexts. Models for particular contexts can then be automatically generated from this global requirements model. As well, the thesis proposes a new class of requirements-about-requirements called awareness requirements. Awareness requirements are naturally operationalized through feedback controllers – the core mechanisms of every adaptive system. The thesis presents an approach for systematically designing monitoring, analysis/diagnosis, and compensation components of a feedback controller, given a set of awareness requirements. Situations requiring adaptation are explicitly captured using contexts.

# Acknowledgments

I would like to express my gratitude to my supervisor Prof. John Mylopoulos for his inspiration, guidance, advice, support, and patience. I feel privileged to have been his Ph.D. student.

I thank Prof. Steve Easterbrook and Prof. Eric Yu for being members of my Ph.D. advisory committee and for their comments, suggestions, and support. I also thank Prof. Pete Sawyer for his participation in the final defense of this thesis as an external examiner and for his helpful comments.

It was Prof. Yves Lespérance at York University who introduced me to the area of goal modeling and requirements engineering, so I thank him for that and for his support and collaboration throughout the years as well as for his helpful comments on this thesis.

I am grateful to my colleagues who worked with me on portions of the research presented in this thesis: Dr. Yijun Yu, Dr. Sotirios Liaskos, and Vítor Souza. I appreciate the discussions, exchange of ideas and other aspects of our collaboration.

I am thankful to my fellow Ph.D. students with whom I crossed paths at the University of Toronto: Neil Ernst, Yiqiao Wang, Rick Salay, and others. I will remember the great discussions we had on goal modeling, requirements, preferences and other topics!

I would also like to thank IBM Canada's Centre for Advanced Studies and Dr. Marin Litoiu personally for providing me with their fellowship for studying adaptive business processes.

Last but not least, I want to thank my parents, my brother, my relatives and friends for their support and encouragement through the years.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1
# Introduction

## 1.1 From Nature to Computer Science

While sometimes used interchangeably in the scientific literature, the terms *variability* and *variation* have different meanings, and that difference is quite important. Variation refers to the actually present differences among the individuals in a population (or some sample thereof). It can be directly observed and is thus relatively easy to measure. On the other hand, variability refers to the potential of a population to vary. Thus, in biology, variability of a trait describes its propensity to change in response to environmental and genetic influences, while genetic variability or diversity measures the actual variation of species in a population. There is, of course, a relationship between variation and variability. If there is a variation in some trait, then it has to be variable. The reverse is not true.

The idea of exploiting variability for adaptation is rooted in nature itself. Looking at the evolution of species in nature as described by Charles Darwin, we can see that variability and variation play crucial roles in that process. Variability is an important factor in evolution as it affects an individual’s response to environmental stress and thus can lead to differential survival of organisms within a population due to natural selection of the fittest variants. Overall, the evolutionary process can be seen as a result of three basic actions: *reproduction, variation,* and *selection*. When entities reproduce, they create a surplus of these entities. Reproduction in nature is not a simple cloning. Thus, variation introduced during the reproduction process amounts to innovation of novel entities [Eib05]. The selection is responsible for promoting the best variants and discarding of the poor ones.

The evolution can be summarized as follows: given a population of individuals, the environment pressure causes natural selection (survival of the fittest), which causes a rise in the fitness of the population over time. It can also be viewed as *adaptation* of the population to the changing environment conditions. An evolutionary process can be viewed as a *generate-and-test search algorithm* regulated by a number of parameters [Eib05]. There are two main forces in the evolutionary process. The first one, *variation*, creates the necessary diversity and thus facilitates novelty through the use of recombination and mutation operations. *Recombination* (crossover) is applied to two or more parents and results in one or more new individuals, the children. Recombination works by exchanging genetic material between parent individuals: the DNA of a child is a combination of the DNA of its parents. This exchange results in the children that are genetically similar, but not identical to their parents. *Mutations* are basically the results of DNA copying errors. They can be viewed as being applied to individuals (children) to produce random small changes. A large number of variant individuals produced in a population increases the chances of survival of the species.

The second force in the process of evolution is *selection*. As opposed to variation, selection reduces diversity in a population. In nature, less fit individuals simply do not survive in a given environment, thus not producing any descendants. On the other hand, individuals that fit their environment well procreate and thus their variation of the genetic material is passed to the new generation. In effect, each particular variation of the genetic material is "tested" throughout the lives of individuals to see whether they are fit to survive in the current environment. The genetic material that does not pass the test (i.e., inside the individuals that die before reproducing) is discarded.

The ideas of evolution and natural selection found their way into computer science and are exemplified by a plethora of evolutionary computing approaches (see [Eib05] for an overview) that try to mimic evolution in nature quite closely. There are many flavours of evolutionary computing approaches that differ in certain aspects, such as the internal representation of the solutions (e.g., tree-based vs. string-based). In effect, these approaches attempt to simulate evolution – the evolution of solutions to problems. They produce a population of variant solutions through recombination and mutation operations (i.e., there is non-determinism involved) and assess these solutions using evaluation functions. Individual solutions with high scores are used with higher probability to create a new generation of individuals, which are in turn evaluated using the functions. The solutions with lower scores are usually discarded. This process stops when a solution with desired properties is found.

One can view this simulated evolution as an optimization (or approximation) process that over its course approaches optimal values. However, as already mentioned, evolution is also frequently seen as a process of adaptation. So, from this point of view, fitness is seen not as an objective function to be optimized, but as an expression of environmental requirements [Eib05].

Variability in software engineering is also important and is mainly used to support software product lines. We review some of the mechanisms that support variability in software systems in Section 2.3.3.

In this thesis, we look at using variability elicited and analyzed during the requirements engineering (RE) phase of the software development process to guide the development of customizable and self-adaptive software systems. The presence of many variant behaviours in software systems allows the selection of the behaviour that is most suited for the stated user preferences at design time, instantiation time, or even at runtime. Here, behaviour variability provides the foundation for system customization. Moreover, the dynamic selection of the behaviour that is most suitable for the current environment conditions and/or changing requirements is the essence of software adaptation.

In our approach, we take some of the ideas from evolution in nature and retain the two major phases of the process: variation and selection. The variation phase amounts to eliciting and representing alternative ways for meeting high-level system requirements. While variability in nature is not always easy to measure, we explicitly represent these alternatives in goal models and the number of the alternatives can be seen as the measure of system variability. Not unlike genetic variability defining the space of possible variations in individuals in a population, variability in our approach provides the space of possible variations of system behaviour. Likewise, the purpose of variability is the same: high number of possible variations increases the chance that there is a variation that meets the requirements of a particular environment. In the selection phase of the process, we evaluate alternatives based on how they meet non-functional requirements and pick the one that achieves the goals of the system while maximizing the satisfaction of these quality requirements. Additional constraints, such as the current context, may also be used in our approach for selecting behaviour alternatives.

## 1.2 Variability in Goal Models

Requirements engineering is a process above all targeted at discovering the intended purpose of a software system. How well the system meets that purpose is the main factor in determining its success [NE00]. This is why RE together with a common goal modeling notation are the starting point for our research.

The purpose of the system is discovered with the help of the RE process through the identification of system stakeholders together with their needs and properties and the modeling of this information in a way that makes various kinds of analyses possible [NE00]. Until relatively recently, requirements were mainly understood as features (e.g., functions, interfaces, properties, etc.) the system-to-be has to possess. However, the trend in RE has shifted toward being more focused on the problem domain and the needs of stakeholders rather than on the solutions to those problems [NE00]. The *problem* can be viewed as stakeholder dissatisfaction with the current state of the domain and a desire for change to eliminate or at least ameliorate perceived problems. A solution describes ways to get to and as well as to maintain the desired state of affairs. A well-known way to depict the problem and the solution was suggested by Michael Jackson [Jac97]. He proposed two overlapping domains: the *machine* and the *world* (the environment of the machine). Requirements reside in the world and describe the desired conditions there. On the other hand, the machine domain includes everything pertaining to the machine. The very subject of software development (or, more broadly, of systems development) is the construction of the machine that interacts with and influences its environment (the world) in such a way so as to meet the desired conditions.

In RE, *goals* (e.g., [DLF93]) have become the notion of choice for elicitation and modeling of the requirements a system has to meet. Therefore, goals are viewed as high-level abstractions of the problem, representing desired states of the world that stakeholders want to reach or maintain. Examples of such functional stakeholder goals are Schedule Meeting and Process Order. The goals may originate (or belong to) different stakeholders [Yu97], thus allowing the description of the problem domain as well as the statement of the expectations for the system-to-be from various points of view. Goals are captured using goal models where they can be refined into sets of subgoals, which, when satisfied together, imply the satisfaction of the original goal. We call this refinement an *AND decomposition* of goals. Similarly, goals can be related to potentially many subgoals that represent the means for achieving them. This refinement is called the *OR decomposition*. The capability to represent alternative ways to attain goals plays a crucial role in this thesis. It allows for capturing the different ways that high-level goals can be gradually refined into solutions. In fact, the quantity of possible alternatives for attaining a top-level goal G grows exponentially with the number of OR decompositions in the subtree representing the refinement of G, as all combination of choices in all OR decompositions need to be considered to enumerate these alternatives. Thus, OR decompositions in goal models allow us to represent *variability* in ways that these goals can be achieved, i.e. the variability in the problem domain. We call these *intentional* OR decompositions or *variation points* (VPs), as they represent *intentional variability* in goal models. In addition to functional goals that are either fulfilled (satisfied) or not fulfilled (denied), there also exist qualitative goals that are hard to define formally and difficult to assess objectively, for example, [Maximize] Customer Satisfaction and Minimal Disturbance [to meeting participants]. These are captured as *softgoals* in our models. A softgoal does not have a clear-cut fulfillment condition and one can only say that a particular softgoal is *satisficed* to an acceptable degree. In our goal models, there are four satisfaction levels for softgoals. They can be fully/partially satisfied or fully/partially denied. A goal/softgoal can also *contribute* positively and negatively to a softgoal, which means that its fulfillment or denial influences the satisficing of the target softgoal. The contribution relationships include help/hurt as well as make/break. Thus, contributions model the effects of the achievement/denial of goals on important quality attributes of the system. Thanks to softgoals, we can differentiate between various alternative goal model refinements that capture ways to fulfill the same functional goals, but with possibly different satisfaction/denial levels for softgoals. There exists a goal reasoning algorithm that can help with the selection of the best alternative for the fulfillment of the root goal given the desired satisfaction level for softgoals [SGM04].

## 1.3 Capturing Domain Variability with Contexts

The analysis of the domain of the system-to-be is an important activity in the requirements engineering process as domain properties and characteristics influence and constrain the space of solutions to a given set of system requirements and also affect the requirements themselves. However, requirements engineering approaches such as KAOS [DLF93] or Tropos [BPGM04] do not emphasize the development of models capturing the properties of domains, their variations, and, most importantly, the effects of these variations on system requirements. While intentional variability in goal models has been addressed by recent research [LLYY06], the problem of capturing the effects of contextual variability on requirements models has not received equal attention. Goal modeling approaches assume that the environment of the system is quite uniform and attempt to elicit and refine system goals in a manner that would fit most problem instances. Thus, domain variability is not addressed in these approaches and its impact on system requirements is mostly ignored. This problem becomes even more pronounced as software systems are being used in diverse, changing environments. Moreover, user expectations of the system are frequently tied to the properties of its current operational environment. This is one of the defining characteristics of the mobile and pervasive computing domains. However, the need to represent and address domain variability in requirements and systems development is not confined to the above domains. For example, in the area of business process management, the ability to adapt to changing environment conditions or user characteristics is also extremely important and may constitute an important competitive advantage. In Chapter 4 of this thesis, we present an approach that allows exploring the effects of domain variability on intentional variability in goal models. There, we first propose a generic formal context model that allows the modeler to specify for any modeling notation under which circumstances various elements of the model (e.g., nodes, links) are *visible* (i.e., present) in the model. This is done by attaching sets of contextual tags to these elements. The tags such as *largeOrder* or *importantCustomer* have definitions referring to domain properties and thus can be used to indicate that, for example, a node is visible in the domain variant characterized by a large order and an important customer. In domains where these tags are not *active* the tagged node will not appear in the model. Contextual tags allow us to decouple context definitions from their effects. Moreover, tag inheritance (including multiple and non-monotonic inheritance) is supported, which helps with structuring the variability in the environment and with incremental development of context-enriched goal models. Overall, contextual tags can be used to specify the effects of domain variability on any type of model. All of the contextual variability can be captured using a single context-enriched model. In this thesis, we further apply the generic context framework to goal models. This allows us to capture in a single goal model the effects of various domain characteristics on requirements for software systems. Then, given a particular version of the domain, a goal model variation can be produced representing the requirements for that particular domain. Capturing the effects of domain variability on system requirements increases the precision and flexibility of requirements models.

Context-awareness, which originates in ubiquitous and pervasive computing, can be considered a type of adaptivity where systems can both sense and react to changes in their environment. The context framework presented in this thesis can serve as the foundation for the goal-driven approach for the elicitation and analysis of adaptation requirements for context-aware systems, including monitoring, analysis, and adaptation. Along with the awareness-requirements approach presented in Chapter 8 of this thesis, these comprise some of the most important components for the overall approach for the elicitation, modeling, and analysis of requirements for adaptation.

Moreover, the context framework can be used not only to capture the *external context* of the system – i.e., things outside of the system that have an effect on it, but also the *internal context*. Internal context consists of things within the system that have an important effect on its behaviour. Internal faults are just one example of what could be considered part of the internal context. In Chapter 8, we capture and refine the goals of feedback controllers that arise in the presence of various errors and faults within the system. These errors/faults are modeled using internal contexts.

## 1.4 Preserving Requirements-Level Variability throughout Design

In many goal-oriented requirements (GORE) approaches, intentional variability is frequently bound at the requirements level (e.g., [BPGM04], [Yu97]). This means that while alternative ways to achieve goals are identified along with their evaluation with respect to non-functional criteria, the choices in variation points are made even before the requirements specification is produced. The aim is to select “the best” way to achieve system goals and to develop the corresponding requirements specification, design, and implementation. The resulting system is designed to have the optimal behaviour for achieving its goals. However, a number of difficulties can emerge with this approach. For example, while the choices in the way goals are achieved may, in fact, be the result of informed and unanimous decisions by the stakeholders at the time they are made, due to the dynamic nature of modern enterprises and the difficult competitive landscapes in most business areas, it is possible that at some later time these decisions will appear suboptimal, if not wrong. Also, frequently, it’s not possible to select the best alternatives for every possible user of the system. Thus, software development effort is usually aimed at solutions that would work for the majority of users/customers or for the most common environment conditions, which may lead to non-standard users (e.g., users with various disabilities) not being accommodated by the systems or to particular properties of the environment being ignored. Another difficulty, which arises when variability in the problem domain is not supported in the solution, is when failures occur. Due to the lack of alternative means for achieving failed system goals, the implemented system will not be capable of easily reconfiguring itself (or of being reconfigured) to circumvent the failure.

To avoid the above-mentioned problems we suggest that while a certain pruning of alternatives represented in goal models can be made based on their current and expected feasibility, the remaining variability elicited in the problem domain and captured using goal models must be preserved in the solution domain, i.e. throughout the subsequent phases of software development. This idea provides the foundation for our requirements-driven approach for designing customizable/configurable and adaptable/adaptive software systems. Using a small set of mappings linking goal model patterns to the appropriate patterns in the target notations as well as by exploiting the hierarchical nature of goal models, we propose a set of variability-preserving processes that given high-variability (HV) goal models capturing a space of alternatives in the problem domain will each be able to produce models that are downstream from requirements in the software development process – at the design and/or implementation level. These transformations are performed after goal models are augmented with the appropriate annotations that capture the aspects of the problem domain, which are required by the transformations, but cannot be represented by the standard goal modeling notation. The idea of goal model annotations is not new. For instance, it was used extensively in [LL06] in the context of the *i** modeling framework [Yu97] precisely for the same purpose – to support the transformation of relatively high-level *i** models into formal specifications.

In addition to preserving requirements-level variability, the choices among the alternatives in the generated models (e.g., in terms of the system behaviour, configuration, etc.) can be linked back to the selections within the appropriate goal model variation points.

While increasing the cost of systems development by requiring that multiple functionally similar behaviours be implemented, the approach provides many benefits since requirements-level variability preservation downstream can be helpful in a number of ways. For designing configurable and customizable systems alternative ways to meet stakeholder goals will lead to alternative system behaviours that, while having the same functionally, may differ in their non-functional characteristics. In this thesis (Chapter 5), we present procedures for systematic generation of the statechart-based high-variability behaviour specifications as well as flexible architectural representations. Those representations are linked back to the goal models' variation points through appropriate parameterization. Thus, the selection of goal model configurations will result in the corresponding behaviour or architecture being automatically selected. This can be done during instantiation of the system variant or potentially at runtime.

In addition, we show that other types of models can be generated from high-variability goal models. The intentional variability characterized using such models is able to help with the creation and maintenance of a product line by capturing the intentions behind the product line and by providing the rationale for the existence of particular product versions as well as for requirements-driven selection of product configurations from the product line. In Section 5.3.1, we discuss the mapping of high-variability goal models to feature models [CE00] that are heavily used in domain engineering and product line engineering. In addition to being able to support the creation of various types of features (optional, alternative, etc.) from appropriately annotated goal models, there is also a possibility to use contribution links among goals to identify useful relationships among features in the generated feature models (e.g., potential conflicts among features). Thus, in the three target modeling notations, which are generated from HV goal models in Chapter 5, intentional variability present in the source goal models is preserved as *configuration variabilit*y in feature models, *behavioural variability* in statecharts, and *structural variability* in connector-component architectural models.

Overall, preserving variability through various stages of the software development process and ultimately in the fully developed system allows the use of goal models as high-level abstractions of the behaviour of the system. This affords the opportunity for doing analysis of the performance of systems as well as for their configuration and reconfiguration at high level, using goal models. Then, configurations selected by analyzing high-variability goal models with the help of goal reasoning techniques ([SGM04], [GMNS02]) can be used to adapt/customize systems at runtime or during their deployment.

## 1.5 Exploiting Variability in Business Process Management

To further test the idea described so far, we looked at the domain of business process (BP) modeling and management. BPs are a good fit for our requirements-driven approach since they are conceptual representations of business activities in organizations and as such can be modeled at high level. Systematic requirements-driven development of business processes is very beneficial to enterprises since in most cases their success depends on how their BPs fit their business practices, how reliable and flexible they are, etc. Using BPs we can represent systems at the level that is accessible to both IT people tasked with the implementation and maintenance of these systems as well as to business users who run the enterprise.

While there are many definitions of what a business process is, in general a BP is seen as a collection of activities that achieves some *purpose* or *objective*. Thus, we can say that business processes specify ways to achieve *business goals*. However, despite this, most leading BP modeling notations and approaches lack support for goals. Rather, they model business processes at the workflow level, in terms of activities, flows, etc. (e.g., [DAH05]). Starting to model BPs at the workflow level is somewhat akin to doing design without doing analysis. Then, the activities within the process are not explicitly linked to requirements and to the business objectives that they are achieving. The rationale behind workflows is not modeled and choices within business processes are modeled at a rather low-level. In addition, business processes within organizations need to be flexible to be able to accommodate changing business priorities, business cases with varying characteristics, etc. Various techniques are employed for changing and configuring BPs (e.g., business rules). However, these are usually also low-level, may require considerable expertise from the business users, and are often not properly linked to business objectives.

In Chapter 6, we offer our method for requirements-driven design and configuration of business processes. It represents a case study of the approach described in Chapter 5 of this thesis. Moreover, in this case study we transform HV goal models into high-variability *executable* models. The approach helps with the above-described difficulties in BP modeling and management by proposing goal-driven method that starts with the identification and refinement of high-level business goals with the help of GORE while emphasizing the capture of variability in the business domain as well as the analysis of alternative ways of attaining business goals using high-variability goal models. Here, business goals, both functional and especially non-functional, can now be explicitly represented during business process development. These goals provide the rationale for business process design so that the activities that are carried out during BP execution can be linked to the goals, whose satisfaction they contribute to. This helps with the maintenance of BPs since business tasks can now be easily traced back to their source requirements.

In our approach, softgoals are used for selecting appropriate alternatives using standard goal model analysis techniques. These goal models are augmented with control flow annotations aimed at capturing the dependencies (or lack thereof) among goals, which cannot be modeled using standard goal models. Then, instead of selecting the best process alternative and implementing it, a semi-automatic variability-preserving process converts the HV goal model representing the goal of a BP and its refinement into an executable business process specification in the WS-BPEL language [BPEL07]. In this mapping process, leaf-level goals are mapped into Web Service invocations, softgoals are not mapped at all since the target language lacks the means to represent non-functional aspects of business processes, while control flow annotations are used to combine the services within a BPEL process in the appropriate manner. Since executable BPEL processes require a lot of implementation details that are not possible to elicit at the requirements level, we only generate skeleton business process specifications in BPEL, while the missing details have to be manually added. As already discussed, the key idea in these types of transformations is the preservation of variability. Here, variation points are represented in the executable BPEL processes using the `switch` constructs with the appropriate parameters that refer back to the original VPs in the goal model as well as to the particular choices made in those variation points. Thus, each goal model configuration has a counterpart in the executable business process specification, which allows the configuration of BPs with the help of goal models. Since the selection of goal model variations is normally guided by softgoals representing important criteria for the processes, using our approach, instances of BPs can be configured at a very high level, in such user-oriented terms as “my preference for this process instance is to maximize performance over minimizing cost”. Therefore, in the approach presented in this thesis, business process configuration is requirements-driven. A prototype infrastructure for BP configuration is also presented. While in the case study we dealt only with BP configuration at the time of instantiation, the approach and most of the infrastructure can be used for runtime reconfiguration of processes. Additional issues such as process consistency will, of course, have to be addressed.

## 1.6 Behaviour Variability as the Foundation for Adaptive Systems Design

Continuing on the topic of the use of requirements-level variability in goal models in the later stages of software development and even at runtime, in Chapter 7 we show that HV designs developed based on HV goal models can become the foundation for the development of adaptive or autonomic [KC03] systems.

The complexity of modern software-intensive systems is growing. Software is becoming truly ubiquitous. Intricate software systems now routinely appear in variety of domains where just a few years ago the complexity of software was orders of magnitude simpler. For instance, shipping containers, delivery trucks and rental cars are now being equipped with geo tracking solutions that are able to provide just-in-time information on their location. This information can be utilized by the owning companies to improve their operations. Televisions, telephones, cars, music players, etc. now have network connectivity and are able to download and play content on varying-bandwidth networks, display targeted advertisements, etc. And the list goes on. There are expectations from users that software systems will integrate seamlessly, collaborate to achieve user goals, adapt to changing priorities/preferences and environment conditions while optimizing their performance. However, together with the complexity of software systems, this increases their management complexity. Self-adaptive or autonomic systems promise to move this complexity from humans into the software itself, thus reducing software maintenance cost, improving performance of systems, customer satisfaction, etc. The main feature of adaptive systems is their ability to adapt to changing requirements and environment conditions, to recover from failures, to lower management cost, and so on.

There are a number of ways to design adaptive systems. For instance, one possibility is to go the route of software agents and multiagent systems (MAS) and integrate planning/reasoning and social abilities into the system (e.g., [SKWL99]). This way, the components of adaptive systems will be able to cooperate, coordinate their actions aimed at achieving their system goals, delegate tasks to external agents, etc. – all to deliver the required functionality of the system. This is a powerful paradigm that can handle dynamic and incompletely known environments, goals that are unknown at design time, etc. However, these capabilities come at the cost of enormous complexity, the need for heavy formalization, the lack of transparency and predictability.

We, on the other hand, propose that high-variability goal models be used as the basis for designing software that supports a space of possible behaviours, all delivering the same functionality. The idea behind this approach is to realize the problem domain variability captured in HV goal models in the isomorphic space of solution domain variability in terms of system behaviour (while preserving the traceability among the alternatives) and to utilize this variability to change the behaviour of systems at runtime with the help of goal models. This method relies on the variability-preserving transformation approach that we discuss in Chapter 5 to systematically generate high-variability software design. In Chapter 7, we argue that adaptive systems must be aware of their goals, ways to achieve them, non-functional constraints on the achievement procedures as well as the domain constraints. This is why goal models are the starting point for the approach. Then, properly augmented goal models capturing the goals of the system, ways to achieve them, evaluations of the various alternatives with respect to non-functional criteria, as well as additional information such as environment assumptions can act as the main source of knowledge for the adaptive system at runtime, guiding it through the process of selecting appropriate behaviour alternatives. We further demonstrate how such adaptive behaviours as self-configuration, self-optimization, etc. can be realized using this approach. It is clear that not all of the alternative system behaviours can be identified at design time. New behaviours can potentially be discovered and implemented by systems at runtime. However, this is a difficult task, which our approach minimizes the need for. Overall, the requirements-based approach for adaptive systems design presented in Chapter 7 provides the basis for designing predictable and transparent systems.

## 1.7 Towards Awareness Requirements

In general, there are a number of ways to operationalize requirements. One possibility for developing systems is to carefully look for domain assumptions and system functionality that together satisfy the requirements, as done in the KAOS approach [DLF93]. Another way is an NFR Framework-like [CNYM00] process-oriented approach where alternatives are identified and selections among these alternatives are systematically made throughout the process based on their evaluations with respect to quality criteria, thus convincing the stakeholders that the functional/non-functional requirements of the system are met. These approaches are better suited for design time since the system is designed to fully cope with achieving its functional and non-functional goals. Another way to attain requirements that is more suited for runtime is to accept the possibility of the system failing to achieve some of its goals while equipping it with the machinery to monitor for their achievement, to analyze the state of the system as well as the state of its environment, and to either retry the achievement procedures for a failed goals or to select alternative means for the goals' attainment.

Software systems supporting many alternative ways of satisfying their goals can change their behaviour to optimize themselves, to recover from failures, to adapt to changing circumstances, etc. While in the previous section we discussed the use of high-variability designs systematically derived from high-variability goal models as the foundation for developing customizable and adaptive systems, we have not explicitly captured the adaptation requirements or explored their origins.

There is a lot of interest in systems that self-adapt to changes in their environments or requirements while continuing to achieve their objectives. Recent research increasingly views such systems as consisting of two distinct parts, one delivering the functionality of the system, while another being a monitor-diagnose/analyze-plan-execute loop that is designed to provide this adaptive behaviour (e.g., [CLGI09], [ICAC09]). Feedback loops [HDPT04] are frequently used to augment the main systems functionality to introduce monitoring, analysis, compensation, etc. activities. Supporting this idea, Brun et al. [BMGG09] argue that "the feedback behaviour of a self-adaptive system, which is realized with its control loops, is a crucial feature and, hence, should be elevated to a first-class entity in its modeling, design, implementation, validation and operation". Similarly, Garlan et al. also advocate for separating the main functionality of the system from its adaptation concern by making the latter external, as opposed to hard-wired into the main system [GCS03]. However, if feedback loops (and control loops in general) are an architectural solution to self-adaptation, then what are the requirements problems that these solutions fulfill? To find the answer to this question we are interested in exploring the requirements that lead to the feedback loop functionality. In Chapter 8, we treat feedback loops as meta-processes designed to run the main system while achieving their requirements. One kind of requirements that lead to feedback loop functionality is *self-awareness requirements* (or simply awareness requirements), the main subject of Chapter 8 of this thesis. Awareness requirements are requirements about the runtime success/failure of other requirements, including qualities and domain assumptions. For example, we may want to state that the system should not allow a certain goal such as Schedule Meeting to fail, that this goal has to have a "high" success rate (perhaps, 95%), that the goal Receive Order should not fail more than twice in one hour, or that the average time to achieve the Supply Customer goal must be no more than two days.

We call these "awareness" requirements because in order to fulfill these requirements-about-requirements the system necessarily needs to be *aware* of how it is doing with respect to achieving its (regular) requirements through some type of monitoring. The system will also need to be able to analyze its performance with respect to these requirements, determine if changes to its behaviour are warranted, and, finally, enact these changes. Thus, awareness requirements can be viewed as a type of meta-requirements that refer to other requirements and help in eliciting adaptation requirements. In Chapter 8, we explore the origins and the significance of this type of requirements in deriving feedback-based adaptive system architecture together with precise adaptation requirements for that architecture. In our approach for handling these types of requirements, we utilize UML with OCL constraints for their representation and analysis. We then propose a systematic process for generating architectural elements consisting of one or possibly more feedback controllers to accommodate a given set of awareness requirements. In addition, we describe the modeling and the refinements of various goals of the feedback controllers (designed to achieve awareness requirements) such as Monitor, Analyze, Adapt, etc. using the familiar goal modeling notation augmented with domain assumptions and contexts. Here, the goals of the controllers can be refined in a similar fashion to regular system goals, complete with the elicitation, modeling, and analysis of variability with respect to quality attributes. For example, there can be various options for finding out whether an instance of a certain goal referred to by an awareness requirement is achieved: through monitoring its post-condition (provided it is monitorable within the system), through getting it from an external system, through asking the human user explicitly, etc. Each of these alternatives may have different costs associated with them as well as different precision. Depending on the priorities among the quality attributes in the model, different options may be preferred by the controller. Similar refinement takes place for other goals of the controller. Additionally, context annotations are used to capture internal contexts of the controller that can represent types of possible failures, their origins and/or their locations. This way, we are able explicitly identify, model, and analyze *at the intentional level* the error handling that is to be performed by the controller. Overall, the approach allows for the elicitation and analysis of adaptation requirements including monitoring, analysis/diagnosis, as well as adaptation/compensation.

Awareness requirements are not the only source of adaptation requirements. For instance, context-awareness is another such source. Similarly, maintenance goals (as described in [DLF93]) are still another source of requirements for controllers as these goals also need to be monitored for violations and their desired conditions need to be restored when necessary. All of these sources of adaptation requirements eventually need to be integrated to derive a complete set of adaptation requirements for adaptive, context-aware software systems. Therefore, the framework for elicitation and analysis of awareness requirements and for the derivation of monitoring, analysis/diagnosis, and adaptation requirements for feedback controllers is a first step in this direction.

## 1.8 Research Question and Research Methodology

The research question that this thesis attempts to answer is the following:

*How can high-variability requirements models be utilized for the systematic design of (and use with) customizable and adaptive systems?*

Each of the key chapters in the thesis can be considered as having its own research question(s). Addressing these sub-questions together provides the answer to the main research question stated above. Here are the research sub-questions for the main chapters of the thesis:

- *How can we capture a new type of variability, domain variability, which was overlooked in the previous research, in requirements models?* (Chapter 4);
- *How do we systematically generate high-variability designs while preserving traceability to requirements?* (Chapter 5);
- *How do we apply the variability-preserving transformation of requirements models to design models to a real-world problem and demonstrate the value of high-level, goal-based systems analysis for software customization?* (Chapter 6);
- *How do we utilize high-variability goal models at runtime for software adaptation?* (Chapter 7);
- *What are some of the sources of adaptation requirements? How do we elicit, analyze, and refine adaptation requirements for self-adaptive systems through the use of awareness requirements?* (Chapter 8).

In terms of the research methodology, we adopt an engineering perspective: we identify concepts and develop methods supported by tools to solve engineering problems. The main chapters of the thesis differ in terms of what mix of concepts, methods, and tools they present. Some are heavy on concepts, while others do not introduce new concepts, but put emphasis on novel methods and tools. Below is the table that presents the concepts, methods, and tools described in the thesis as well as specifies the objectives of the main thesis chapters.

Table 1-1. Concepts, methods, and tools described in the thesis and chapter objectives.

| | Chapter Objectives | Concepts | Methods For... | Tools |
|---|---|---|---|---|
| Chapter 4 | Develop a framework to capture the effects of domain variability on software requirements. | Contextual Tags | Calculating element visibility in context;<br>Capturing domain variability in goal models;<br>Analysis of goal models in context. | Future work |
| Chapter 5,<br>Chapter 6 | Develop methods for systematically addressing problem domain variability in the solution domain. | | Pattern-based variability-preserving mapping to design-level notations;<br>Traceability to requirements;<br>Use of goal models for the analysis of software systems. | Semi-automated mapping to target notations;<br>BP configuration infrastructure;<br>Preference specification tool. |
| Chapter 7 | Develop methods for utilizing high-variability goal models for the development and effective use of customizable and adaptive systems. | | High-variability designs as the basis for adaptive systems;<br>Goal models as the source of knowledge for adaptive systems;<br>Goal models as system abstractions. | Requirements-driven (re-)configuration tool. |
| Chapter 8 | Identify concepts and develop methods for elicitation and analysis of requirements about requirements leading to a novel approach for RE for adaptive systems. | Awareness require-ments | Elicitation, formal specification, and refinement of awareness requirements;<br>Specification, analysis, and refinement of adaptivity requirements derived from awareness requirements. | Future work. |

## 1.9 Structure of the Thesis

The rest of this thesis is structured as follows. Chapter 2 discusses related work relevant to this thesis. Chapter 3 describes the goal modeling notation that we use throughout the thesis. In Chapter 4, we argue that domain variability modeled by contexts is an important source of influence on requirements and must be captured and analyzed during requirements analysis. We propose a context modeling framework while illustrating its use with goal models. This chapter is based on [LM09]. Chapter 5 presents our approach for generating high-variability solution-oriented models from high-variability goal models. This chapter is based on [YLLM08b]. Chapter 6 presents a case study aimed at evaluating the usefulness of the ideas from Chapter 5. It discusses requirements-driven design and configuration of business processes through the use of variability-preserving transformations of goal models into executable business process specifications. It is based on [LYM07]. Chapter 7 argues that high-variability goal models together with the derived high-variability software designs can be used as the basis for designing adaptive/autonomic systems. This work is based on [LLMY05] and [LYLM06]. Chapter 8 discusses awareness requirements, the requirements about success/failure of other requirements and how they can be used to derive detailed requirements specifications for controller processes in adaptive systems. Chapter 9 offers conclusions and research directions for continuing/extending the work presented in this thesis.

## 1.10 List of Publications

This section presents the list of publications that are the basis for portions of this thesis.

| | |
|---|---|
| [LLMY05] | A. Lapouchnian, S. Liaskos, J. Mylopoulos, Y. Yu. Towards Requirements-Driven Autonomic Systems Design. In Proc. *ICSE 2005 Workshop on Design and Evolution of Autonomic Application Software (DEAS 2005)*, St. Louis, Missouri, USA, May 21, 2005. *ACM SIGSOFT Software Engineering Notes,* 30(4), July 2005. |
| [LM09] | A. Lapouchnian and J. Mylopoulos. Modeling Domain Variability in Requirements Engineering with Contexts. In Proc. *28th International Conference on Conceptual Modeling (ER 2009),* Gramado, Brazil, Nov 9-12, 2009. A.H.F. Laender, S. Castano, U. Dayal, F. Casati, J.P.M. de Oliveira (Eds.), LNCS Vol. 5829, pp. 115–130, Springer-Verlag, Berlin Heidelberg, 2009. |
| [LYLM06] | A. Lapouchnian, Y. Yu, S. Liaskos, J. Mylopoulos. Requirements-Driven Design of Autonomic Application Software. In Proc. *16th Annual International Conference on Computer Science and Software Engineering CASCON 2006*, Toronto, Canada, Oct 16–19, 2006. |

[LYM07] A. Lapouchnian, Y. Yu, J. Mylopoulos. Requirements-Driven Design and Configuration Management of Business Processes. In Proc. *5th International Conference on Business Process Management (BPM 2007)*, Brisbane, Australia, Sep 24-28, 2007. G. Alonso, P. Dadam, and M. Rosemann (Eds.), LNCS Vol. 4714, pp. 246–261, Springer-Verlag, Berlin Heidelberg, 2007.

[YLLM05] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos. Requirements-Driven Configuration of Software Systems. In Proc. *WCRE 2005 Workshop on Reverse Engineering to Requirements (RETR'05)*, Pittsburgh, PA, USA, November 7, 2005.

[YLLM08a] Y. Yu, A. Lapouchnian, J.C.S.P. Leite, J. Mylopoulos. Configuring Features with Stakeholder Goals. In Proc. *23rd Annual ACM Symposium on Applied Computing (SAC 2008)*, RE Track, Brazil, March 16-20, 2008.

[YLLM08b] Y. Yu, A. Lapouchnian, S. Liaskos J. Mylopoulos, J.C.S.P. Leite. From Goals to High-Variability Software Design. In Proc. *17th International Symposium on Methodologies for Intelligent Systems (ISMIS 2008)*, Toronto, Canada, May 20-23, 2008. *Invited Paper*. A. An, et al. (Eds.), Foundations of Intelligent Systems, ISMIS 2008, LNAI Vol. 4994, pp. 1–16, Springer-Verlag, Berlin Heidelberg, 2008.

# Chapter 2
# Related Work

## 2.1 Requirements Engineering

The main measure of success for a software system is the degree to which it meets its purpose. Therefore, identifying this purpose must be one of the main activities in the development of software systems. It has been long recognized that inadequate, incomplete, ambiguous, or inconsistent requirements have a significant impact on the quality of software. Thus, Requirements Engineering (RE), a branch of software engineering that deals with elicitation, refinement, analysis, etc. of software systems requirements gained a lot of attention in academia as well as in industry.

There are many definitions of requirements engineering. For example, Zave [Zav97] defines it as "the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and their evolution over time and across software families". van Lamsweerde [Lam00] describes the following intertwined activities that are covered by requirements engineering (a similar set of activities is also featured in [NE00]):

- *Domain analysis*: the environment for the system-to-be is studied. The relevant stakeholders are identified and interviewed. Problems with the current system are discovered and opportunities for improvement are investigated. Objectives for the target system are identified.
- *Elicitation*: alternative models for the target system are analyzed to meet the identified objectives. Requirements and assumptions on components of such models are identified.
- *Negotiation and agreement*: alternative requirements and assumptions are evaluated; risks are analyzed by the stakeholders; the best alternatives are selected.
- *Specification*: requirements and assumptions are formulated precisely.
- *Specification analysis*: the specifications are checked for problems such as incompleteness, inconsistency, etc. and for feasibility.

- *Documentation*: various decisions made during the requirements engineering process are documented together with the underlying rationale and assumptions.
- *Evolution*: requirements are modified to accommodate corrections, environmental changes, or new objectives.

Modeling appears to be a core process in RE. An existing system/organization as well as the possible alternative configurations for the system-to-be are typically modeled. These models serve as a basic common interface to the various activities described above. Requirements modeling is the process of "building abstract descriptions of the requirements that are amenable to interpretation" [NE00]. Modeling facilitates requirements elicitation by guiding it and helping the requirements engineer look at the domain systematically. Models also provide a basis for requirements documentation and evolution. While informal models are analyzed by humans, formal models of system requirements allow for precise analysis by both the software tools and humans. Such translation from informal needs of stakeholders to formal specifications is one of the main difficulties in RE.

Despite the fact that the idea of requirements had been around for quite some time, a number of important properties of software requirements remained unknown until mid-1990s. In [Jac97], Michael Jackson makes the distinction between the *machine*, which is one or more computers that behave in a way that satisfies a given set of requirements with the help of software, and the *environment*, which is the part of the world with which the machine will interact and in which the effects of the machine will be observed. When the machine is put into its environment, it can influence that environment and be influenced by that environment only because they have some *shared phenomena* in common. That is, there are events that are common to both. Requirements exist in the environment. They are conditions over the events and states of the environment (or, possibly, the events and states in the shared phenomena) and can be formulated in a language accessible to stakeholders. So, the requirements can be stated without referring to the machine. In order to satisfy its requirements the machine acts on the shared phenomena. Through the properties of the environment (causal chains) it can also indirectly affect the private phenomena of the environment through which the requirements are expressed. Therefore, the description of the requirements must describe the *desired* (optative) conditions over the environment phenomena (*requirements*) and also the *given* (indicative) properties of the environment (*domain* *properties*) that allow the machine participating only in the shared phenomena to ensure that the requirements are satisfied in the environment phenomena [Jac97] (Parnas and Madey [PM95] independently made the same distinction between requirements and domain properties). Another distinction made by Jackson as well as by Parnas and Madey was between the requirements and software *specifications*, which are formulated in terms of the machine phenomena in the language accessible to software developers. Further, another important distinction must be made about requirements and environment *assumptions* (sometimes called *expectations*). Even though they are both optative, requirements are to be enforced by the software, while assumptions are to be enforced by agents in the environment. The assumptions specify what the system expects of its environment. For example, one could have an assumption about the behaviour of a human operator of the system.

Requirements engineering is generally viewed as a process containing two phases. The *early requirements phase* concentrates on the analysis and modeling of the environment for the system-to-be, the organizational context, the stakeholders, their objectives and their relationships. Understanding the needs and motivations of stakeholders and analyzing their complex social relationships helps in coming up with the correct requirements for the system-to-be. Such models are reference points for the system requirements and can be used to support the evolution of requirements that stems from changes in the organizational context of the system. The *late requirements phase* is concerned with modeling the system together with its environment. The system is embedded into the organization; the boundaries of the system and its environment are identified and adjusted, if needed; the system requirements and assumptions about the environment are identified. The boundary between the system and its environment is initially not well defined. The analyst will try to determine the best configuration of the system and its environment to reliably achieve the goals of the stakeholders. Putting too much functionality into the system could make it, for example, too complex and hard to maintain and evolve, while making too many assumptions about the environment may be unrealistic.

Within the realm of requirements there exists another important division, *functional* versus *non-functional* requirements. Functional requirements specify the functions or services of the system. On the other hand, non-functional (quality) requirements (NFRs) represent software system qualities (e.g., security, ease of use, maintainability, etc.) or properties of the system as a whole.

NFRs are generally more difficult to express in an objective and measurable way. Thus, their analysis is also more difficult.

## 2.2 Goal-Oriented Requirements Engineering

In recent years, the popularity of Goal-Oriented Requirements Engineering approaches has increased dramatically. The main reason for this is the inadequacy of traditional systems analysis approaches (e.g., [DeM78], [Ros77], [RBPE91]) when dealing with more and more complex software systems. At the requirements level, these approaches treat requirements as consisting only of processes and data and do not capture the rationale for the software systems, thus making it difficult to understand requirements with respect to some high-level concerns in the problem domain. Most techniques focus on modeling and specification of the software alone. Therefore, they lack support for reasoning about the *composite system* comprised of the system-to-be and its environment. However, incorrect assumptions about the environment of a software system are known to be responsible for many errors in requirements specifications [LL03]. Non-functional requirements are also generally described only informally in requirements specifications. Additionally, traditional modeling and analysis techniques do not allow alternative system configurations where more or less functionality is automated, different assignments of responsibility are explored, or otherwise different goal refinements are selected etc. to be represented and compared. Goal-oriented requirements engineering attempts to solve these and other important problems.

It is important to note that a goal-oriented requirements elaboration process ends where most traditional specification techniques would start [LL03]. Overall, GORE focuses on the activities that precede the formulation of software system requirements. The following main activities are normally present in GORE approaches: goal elicitation, goal refinement and various types of goal analysis, and the assignment of responsibility for goals to agents.

Most of early RE research concentrated on what the software system should do and how it should do it. This amounted to producing and refining fairly low-level requirements on data, operations, etc. Even though the need to capture the rationale for the system under development was evident from the early definitions of requirements (e.g., “requirements definition must say why a system is needed” [RS77]), little attention was given in the RE literature to understanding why, indeed, the system was needed and whether the requirements specification really captured the needs of the stakeholders. Not enough emphasis was put on understanding the organizational context for the new system. In general, the tendency in the RE modeling research has been to abstract the low-level programming constructs to the level of requirements rather than pushing requirements abstractions down to the design level [BPGM04]. This explains why the stakeholders with their needs and the rest of the social context for the system could not be adequately captured by requirements models.

*Goals* have long been used in artificial intelligence (AI)(e.g., [Nil71]). Yet, Yue was likely the first one to show that goals modeled explicitly in requirements models provide a criterion for requirements completeness [Yue87]. A model of human organizations that views humans and organizations as goal-seeking entities can be seen as the basis for goal modeling. This model is dominant in the information systems field [CH98]. However, it remains rather implicit in the RE literature.

There are a number of goal definitions in the current RE literature. For example, van Lamsweerde [Lam00] defines a goal as an objective that the system should achieve through cooperation of agents in the software-to-be and in the environment. Anton et al. [AMP94] state that goals are high-level objectives of the business, organization or system; they capture the reasons why a system is needed and guide decisions at various levels within the enterprise.

An important aspect of requirements engineering is the analysis of non-functional (quality) requirements (NFRs) [MCN92], [CNYM00]. NFRs are usually represented in requirements engineering models by *softgoals*. There is no clear-cut satisfaction condition for a softgoal. Softgoals are related to the notion of *satisficing* [Sim81]. Unlike regular goals, softgoals can seldom be said to be accomplished or satisfied. For softgoals, one needs to find solutions that are “good enough”, where softgoals are satisficed to a sufficient degree. High-level non-functional requirements are abundant in organizations and quite frequently the success of systems depends on the satisficing of their non-functional requirements.

Goal-oriented requirements engineering views the system-to-be and its environment as a collection of active components called *agents*. Active components may restrict their behaviour to ensure the constraints, which they are assigned, are satisfied. These components are humans playing certain roles, devices, and software. As opposed to passive ones, active components have a choice of behaviour. In GORE, agents are assigned responsibility for achieving goals. A *requirement* is a goal whose achievement is the responsibility of a single software agent, while an *assumption* is a goal whose achievement is delegated to a single agent in the environment. Unlike requirements, expectations cannot be enforced by the software-to-be and will hopefully be satisfied thanks to organizational norms and regulations [Lam01]. In fact, requirements “implement” goals much the same way as programs implement design specifications [LL03]. Agent-based reasoning is central to requirements engineering since the assignment of responsibilities for goals and constraints among agents in the software-to-be and in the environment is the main outcome of the RE process [Lam00].

### 2.2.1 Benefits of Goal Orientation

There are a number of important benefits associated with explicit modeling, refinement, and analysis of goals (mostly adapted from [Lam01]):

- It is important to note that GORE takes a wider system engineering perspective compared to the traditional RE methods: goals are prescriptive assertions that should hold in the system made of the software-to-be *and its environment*; domain properties and expectations about the environment are explicitly captured during the requirements elaboration process, in addition to the usual software requirements specifications. Also, goals provide rationale for requirements that operationalize them. Thus, one of the main benefits of goal-oriented requirements engineering is the added support for the early requirements analysis.
- Goals provide a precise criterion for *completeness* of a requirements specification. The specification is complete with respect to a set of goals if all the goals can be proven to be achieved from the specification and the properties known about the domain [Yue87].
- Goals provide a precise criterion for requirements *pertinence*. A requirement is pertinent with respect to a set of goals in the domain if its specification is used in the proof of at least one goal [Yue87]. Even without the use of formal analysis methods, one can easily see with the help of goal models whether a particular goal, in fact, contributes to some high-level stakeholder goal.

- A goal refinement tree provides *traceability* links from high-level strategic objectives to low-level technical requirements.
- Goal modeling provides a natural mechanism for structuring complex requirements documents [Lam01].
- One of the concerns of RE is the management of conflicts among multiple *viewpoints* [Eas93]. Goals can be used to provide the basis for the detection and management of conflicts among requirements [Lam96], [Rob89].
- A single goal model can capture *variability* in the problem domain through the use of alternative goal refinements and alternative assignment of responsibilities [LLYY06]. Quantitative and qualitative analyses of these alternatives are possible.
- Goal models provide an excellent way to communicate requirements to customers. Goal refinements offer the right level of abstraction to involve decision makers for validating choices being made among alternatives and for suggesting other alternatives.
- Separating stable from volatile information is also important in requirements engineering. A number of researches point out that goals are much more stable than lower-level concepts like requirements or operations [AMP94], [Lam01]. A requirement represents one particular way of achieving some goal. Thus, the requirement is more likely to evolve towards a different way of achieving that same goal than the goal itself. In general, the higher level the goal is the more stable it is.

### 2.2.2 The Main GORE Approaches

In this section, we describe the core of the main GORE approaches.

#### 2.2.2.1 The NFR Framework

The NFR framework was proposed in [MCN92] and further developed in [CNYM00]. The NFR framework (as it is evident from its name) concentrates on the modeling and analysis of non-functional requirements. The goal of the framework is to put non-functional requirements foremost in developer’s mind [CNYM00]. The framework aims at dealing with the following main activities: capturing NFRs for the domain of interest, decomposing NFRs, identifying possible NFR operationalizations (design alternatives for meeting NFRs), dealing with ambiguities, trade-offs, priorities, and interdependencies among NFRs, selecting operationalizations, supporting decisions with design rationale, and evaluating impact of decisions. The main idea of the approach is to systematically model and refine non-functional requirements and to expose positive and negative influences of different alternatives on these requirements.

The framework supports three types of softgoals. *NFR softgoals* represent non-functional requirements to be considered; *operationalizing softgoals* model lower-level techniques for satisficing NFR softgoals; *claim softgoals* allow the analyst to record design rationale for softgoal refinements, softgoal prioritizations, softgoal contributions, etc. Softgoals can be refined using AND or OR refinements with obvious semantics. Also, softgoal interdependencies can be captured with positive ("+") or negative ("–") contributions.

The main modeling tool that the framework provides is the *softgoal interdependency graph* (SIG). The graphs can graphically represent softgoals, softgoal refinements (AND/OR), softgoal contributions (positive/negative), softgoal operationalizations and claims. As softgoals are being refinement, the developer will eventually reach some softgoals (in fact, operationalizing softgoals) which are sufficiently detailed and cannot be refined further. The developer can accept or reject them as part of the target system. By choosing alternative combinations of the leaf-level softgoals and using the provided label propagation algorithm, the developer can see if the selected alternative is good enough to satisfice the high-level non-functional requirements for the system. The algorithm works its way up the graph starting from the decisions made by the developer. The labelling procedure works towards the top of the graph determining the impact of the decision on higher-level goals. It takes into consideration the labels on softgoal refinement links. For example, if a softgoal receives contributions from a number of other softgoals, then the results of contributions of each offspring are combined to get the overall contribution for the parent softgoal. By analyzing these alternative operationalizations, the developer will select the one that best meets high-level quality requirements for the system. The set of selected leaf level softgoals/operationalizations can be implemented in the software. The NFR framework also supports cataloguing design knowledge into three main types of catalogues.

Overall, this framework provides a *process-oriented* approach for dealing with non-functional requirements. Here, instead of evaluation the final product with respect to whether it meets its non-functional requirements, the “emphasis is on trying to rationalize the development process in terms of non-functional requirements” [MCN92]. The provided catalogues are an important aid for a requirements engineer.

In the NFR framework, goals can be refined along two dimensions, their *type* (e.g., security or integrity) and their *topic* (subject matter), such as “bank account”. The framework provides a catalogue of methods for systematic refinement of softgoals. A qualitative label propagation algorithm is used for analysis. The idea of goal parameters is used in this thesis when more details about goals need to be captured (e.g., in Chapter 6) for subsequent generation of workflow-level process specifications.

#### 2.2.2.2 *i**/Tropos

*i** [Yu97] is an agent-oriented modeling framework that can be used for requirements engineering, business process reengineering, organizational impact analysis, and software process modeling. Since we are most interested in the application of the framework to modeling systems’ requirements, our description of *i** is geared towards requirements engineering. The framework has two main components: the Strategic Dependency (SD) model and the Strategic Rationale (SR) model.

Since *i** supports the modeling activities that take place before the system requirements are formulated, it can be used for both the early and late phases of the requirements engineering process. During the early requirements phase, the *i** framework is used to model the environment of the system-to-be. It facilitates the analysis of the domain by allowing the modeler to diagrammatically represent the stakeholders of the system, their objectives, and their relationships. The analyst can therefore visualize the current processes in the organization and examine the rationale behind these processes. The *i** models developed at this stage help in understanding why a new system is needed. During the late requirements phase, the *i** models are used to propose the new system configurations and the new processes and evaluate them based on how well they meet the functional and non-functional needs of the users.

*i** centers on the notion of *intentional actor* and *intentional dependency*. The actors are described in their organizational setting and have attributes such as goals, abilities, beliefs, and commitments. In *i** models, an actor depends on other actors for the achievement of its goals, the execution of tasks, and the supply of resources, which it cannot achieve, execute, and obtain by itself, or not as cheaply, efficiently, etc. Therefore, each actor can use various opportunities to achieve more by depending on other actors. At the same time, the actor becomes vulnerable if the actors it depends upon do not deliver. Actors are seen as being *strategic* in the sense that they are concerned with the achievement of their objectives and strive to find a balance between their opportunities and vulnerabilities. The actors are used to represent the system's stakeholders as well as the agents of the system-to-be.

Actors can be agents, roles, and positions. *Agents* are concrete actors, systems or humans, with specific capabilities. A *role* is an abstract actor embodying expectations and responsibilities. A *position* is a set of *socially recognized* roles typically played by one agent. This division is especially useful when analyzing the social context for software systems.

Dependencies between actors are identified as *intentional* if they appear as a result of agents pursuing their goals. There are four types of dependencies in *i**. They are classified based on the subject of the dependency: *goal*, *softgoal*, *task*, and *resource*.

A Strategic Dependency model is a network of dependency relationships among actors. The SD model captures the intentionality of the processes in the organization, what is important to its participants, while abstracting over all other details. During the late requirements analysis phase, SD models are used to analyze the changes in the organization due to the introduction of the system-to-be. The model allows for the analysis of the direct or indirect dependencies of each actor and exploration of the opportunities and vulnerabilities of actors (analysis of chains of dependencies emanating from actor nodes is helpful for vulnerability analysis).

Strategic Rationale models are used to explore the rationale behind the processes in systems and organizations. In SR models, the rationale behind process configurations can be explicitly described, in terms of process elements, such as goals, softgoals, tasks, and resources, and relationships among them. The model provides a lower-level abstraction to represent the intentional aspects of organizations and systems: while the SD model only looked at the external relationships among actors, the SR model provides the capability to analyze in great detail the internal processes within each actor. The model allows for deeper understanding of what each actor’s needs are and how these needs are met; it also enables the analyst to assess possible alternatives in the definition of the processes to better address the concerns of the actors.

The SR process elements are related by two types of links: *decomposition* links, and *means-ends* links. These links are used to model AND and OR decompositions of process elements respectively. Means-ends links are mostly used with goals and specify alternative ways to achieve them. Thus, this is the way to represent variability in the problem domain in this approach. Thus, the model is able to represent the whole space of alternatives for achieving top-level goals. Decomposition links connect a goal/task with its components (subtasks, softgoals, etc.) A softgoal, a goal, or a task can also be related to other softgoals with softgoal contribution links that are similar to the ones in the NFR framework. The links specify two levels of positive (“+” and “++”) and negative (“–“ and “--“) contributions to the softgoals from satisficing a softgoal, achieving a goal, or executing a task. Softgoals are used as selection criteria for choosing the alternative process configuration that best meets the non-functional requirements of the system. It is possible to link a process element from one actor with an intentional dependency going to another actor to represent its delegation to that actor. The SR model is strategic in that its elements are included only if they are considered important enough to affect the achievement of some goal. The same rule also helps with requirements pertinence. The *i** meta-framework describing the semantics and constraints of *i** is described in the language Telos [MBJK90]. This language allows for the various types of analysis of *i** models (e.g., consistency checking between models) to be performed.

The *i** modeling framework is the basis for Tropos, a requirements-driven agent-oriented development methodology [BPGM04]. The Tropos methodology guides the development of agent-based systems from the early requirements analysis through architectural design and detailed design to the implementation. Tropos uses the *i** modeling framework to represent and reason about requirements and system configuration choices. Tropos has an associated formal specification language called Formal Tropos [FPMT01] for adding constraints, invariants, pre- and post-conditions capturing more of the subject domain’s semantics to the graphical models in the *i** notation. These models can be validated by model-checking.

There are approaches that propose to use *i** with other notations support architectural design. For instance, [LY01] proposes to use the *i**-based Goal-oriented Requirements Language (GRL) together with Use Case Maps [BC96], which provide a way to visualize scenarios using *scenario paths* and superimpose them on models representing the structure of abstract components. In this approach, GRL is used to support goal and agent-oriented modeling and reasoning and to guide the architectural design process.  UCM notation is used to express the architectural design at each stage of the development. GRL provides support for reasoning about scenarios by establishing correspondences between intentional elements in GRL and functional components and *responsibilities* (tasks) in UCM. The iterative process uses GRL to reason about how architectural decisions affect high-level non-functional requirements (using the usual label propagation algorithm), while UCM is used to generate and analyze how responsibilities can be bound to architectural components and the effects of these bindings. As new architectural alternatives are proposed during the UCM analysis, they are analyzed in GRL with respect to their contributions to the quality requirements. It is interesting to note that in the this approach, GRL models have intentional elements that do not model goals/tasks of the actors that are part of the combined system, but represent possible architectural decisions (e.g., put component A into device B) and their impact on non-functional requirements.

#### 2.2.2.3 KAOS

The KAOS methodology is a goal-oriented requirements engineering approach with a rich set of formal analysis techniques. KAOS stands for Knowledge Acquisition in autOmated Specification [DLF93] or Keep All Objects Satisfied [LL03]. KAOS is described in [LL03] as a multi-paradigm framework that allows to combine different levels of expression and reasoning: semi-formal for modeling and structuring goals, qualitative for selection among the alternatives, and formal, when needed, for more accurate reasoning. Thus, the KAOS language combines semantic nets [BL85] for conceptual modeling of goals, assumptions, agents, objects, and operations in the system, and linear-time temporal logic for the specification of goals and objects, as well as state-base specifications for operations. In general, each construct in the KAOS language has a two-level structure: the outer graphical semantic layer where the concept is declared together with its attributes and relationships to other concepts, and the inner formal layer for formally defining the concept. The ontology of KAOS includes *objects*, which are things of interest in the composite system whose instances may evolve from state to state. Objects can be *entities*, *relationships*, or *events*. *Operations* are input-output relations over objects. Operation applications define state transitions. Operations are declared by signatures over objects and have pre-, post-, and trigger conditions. KAOS makes a distinction between *domain* pre-/post-conditions for an operation and *desired* pre-/post-conditions for it. The former are indicative and describe what an application of the operation means in the domain (without any prescription as to when the operation must or must not be applied) while the latter are optative and capture additional strengthening of the conditions to ensure that the goals are met [LL02].

An *agent* is a kind of object that acts as a processor for operations. Agents are active components that can be humans, devices, software, etc. Agents perform operations that are allocated to them. KAOS lets analysts specify which objects are observable or controllable by agents.

A *goal* in KAOS is defined in [Lam04] as a "prescriptive statement of intent about some system whose satisfaction in general requires the cooperation of some of the agents forming that system". Goals in KAOS may refer to services (functional goals) or to quality of services (non-functional goals). In KAOS, goals are organized in the usual AND/OR refinement-abstraction hierarchies. Goal refinement ends when every subgoal is *realizable* by some individual agent assigned to it. That means the goal must be expressible in terms of conditions that are monitorable and controllable by the agent. The requirement and expectation in KAOS are defined in the usual way – the former being a goal under the responsibility of an agent in the system-to-be and the latter being a goal under the responsibility of an agent in the environment. Goal definition patterns are used for lightweight specification of goals at the modeling layer. These are specified in temporal logic and include patterns such as *achieve*, *cease*, *maintain*, *optimize*, and *avoid*. Achieve and cease goals generate behaviours; maintain and avoid goals restrict behaviours; optimize goals are said to compare behaviours [DLF93]. KAOS also has supports additional types of goals [DLF93], such as *satisfaction* goals, *accuracy* goals, etc. Overall, a KAOS specification is a collection of the following core models:

- *goal model* where goals are represented, and assigned to agents,
- *object model*, which is a UML model that can be derived from formal specifications of goals since they refer to objects or their properties,
- *operation model*, which defines various services to be provided by software agents.

KAOS does not provide a method for evaluating the impact of design decisions on non-functional requirements. Overall, KAOS is a well-developed methodology for goal-oriented requirements analysis that is supplied with solid formal framework. During goal refinement, goal operationalization, obstacle analysis and mitigation, KAOS relies heavily on formal refinement patters that are proven once and for all. Therefore, at every pattern application the user gets an instantiated proof of correctness of the refinement for free.

In the context of this approach, van Lamsweerde [Lam03a] proposes a method for software architecture design based on KAOS goal models. The starting point for the method is the software specification that can be systematically produced from software requirements in KAOS [LL02]. Then, components are created for agents that are assigned to achieving goals of the system-to-be. Component interfaces are derived based on the sets of variables the agents control and monitor. A dataflow connector is created for each combination of two components where one controls a variable, which the other agent monitors. This produces an initial architectural view for the system. This model is then restructured according to the desired architectural style. Another pattern-based restructuring follows. This time the aim is to improve the quality of service goals. The interesting aspect of this approach is that it generates the initial architectural model directly from the goal model, using the precedence relationships among goals to create data connectors. This means that the initial software architecture can be automatically generated from a goal model.

## 2.3 Variability Modeling and Analysis

### 2.3.1 Variability in Requirements

One of the major ideas of this thesis is the use of the variability elicited and analyzed during requirements analysis (the variability in the way stakeholder goals can be attained), which is the variability in the problem domain, to systematically derive and utilize variability in terms of alternative software configurations and/or behaviours, i.e., the variability in the solution space. In this section, we review some of the approaches for modeling variability both at the level of requirements and at the level of system design or implementation.

Coming up with a solid (complete, correct, etc.) requirements specification for a software system involves the identification of the many alternative ways to achieve goals, assign goals to agents, draw a system-environment boundary, and so on, as well as making the choice among the identified alternatives. Various GORE approaches support this activity differently. For example, one approach, the Goals-Skills-Preferences framework of [HLM03] aims at designing highly-customizable software systems by discovering as much variability in the problem domain as possible. Here, given high-level user goals the framework attempts to identify *all* the possible ways these goals can be achieved by the combined system. The variability is modeled by the OR decompositions of goals in the usual AND/OR goal graph. A ranking algorithm for selecting the best system configuration among this vast space of alternatives is proposed. It takes into consideration user *preferences*, which are modeled as softgoals and are accompanied by the contribution links relating them to the functional goals, and user *skills*, which are hard constraints on the leaf-level goals. Each user is defined through a skills profile – an evaluation of the user's skills such as vision, speech production, etc. Each leaf-level goal, such as "dictate a letter", needs a certain set of skills from the user to be achieved. It is clear, that the "dictate a letter" goal needs the skill of speech production at a reasonable level. So, the for each particular user, the algorithm can prune the alternatives that are incompatible with the user's skills and then select the best one from the remaining set according to the user preferences. The result is a specification that is tailored for a concrete user.

In Tropos and other *i**-based approaches, variability is captured using means-ends or OR decompositions. The modeling of the alternative responsibility assignments for goals, tasks, softgoals, and resources is also supported. For instance, one can model a number of alternative ways of achieving a certain goal. One is to let the actor that owns the goal achieve it, another is to delegate the goal to another actor through an intentional goal dependency, and the third may be to delegate it to a yet another actor. The same approach can be used to analyze the system-environment boundary. By delegating more goals to the actors who are part of the system-to-be we are modeling the situation where more functionality of the combined system is automated. On the other hand, delegating the achievement of more goals to the actors in the environment will shift the balance in an opposite direction. These alternatives can be analyzed through their contributions to the high-level quality requirements modeled as softgoals. The standard softgoal analysis method based on the NFR framework is used.

In KAOS, variability can be represented using OR goal decompositions and alternative responsibility assignments of goals to agents. Additionally, identified obstacles can be resolved differently by alternative decompositions and application of alternative obstacle resolution patterns. Similarly, various heuristics can be applied to conflict resolution in KAOS. All of these activities can result in alternative system proposals. While KAOS does not provide any integrated tools for qualitative analysis of different alternatives with respect to quality requirements, the use of the NFR framework is often suggested. A quantitative approach that can be to reasoning about system alternatives is proposed in [LL04].

A scenario is defined in [LW98] as a temporal sequence of interactions among different agents in the restricted context of achieving some implicit purpose. Variability analysis gained attention from the scenario-based approaches to software engineering. For example, Sutcliffe et al. [SMMM98] describe collections of scenarios where certain events may alternate and/or appear in various orders. The approach then uses constraints to restrict the possible scenario space to the ones of interest to requirements engineers.

There are a number of approaches the employ scenarios as well as goals. For Potts [Pot95], scenarios are derived from the description of the system's and the user's goals, and the potential obstacles that block those goals. He proposes a scenario schema and a method for deriving a set of *salient* scenarios, scenarios that have a point and help people understand the exemplified system. The idea of scenario salience is thus to limit the number of generated scenarios. The goal of the approach is to use scenarios to understand user needs. According to Potts, this process must precede the writing of a specification for a system. In the proposed scenario schema, each *episode* (a major chunk of activity in a scenario) corresponds directly to a domain goal. The goal-scenario process of Potts is interleaving scenario derivation and goal refinement. Scenarios can be obtained by analyzing goals, goal allocations, obstacles, etc. On the other hand, scenarios may give the analyst an insight about goals and goal refinement. Potts suggests that salient scenarios are beneficial in that they can be used to compare alternative system proposals, including more or less automation, various alternative ways of achieving goals, etc. Also, scenarios can be employed to help in mitigating the effects of obstacles. While it is conceivable to assume that scenarios generated for several system alternatives could occasionally illustrate the benefit of one alternative over another, this approach for evaluation of alternatives does not seem systematic compared to, say, the NFR framework.

Another goal-scenario approach is described by Rolland et al. [RSB98]. The CREWS-L'Ecritoire creates a bi-directional coupling between goals and scenarios. The proposed process has two parts. When a when a goal is discovered, a scenario can be authored for it and once a scenario has been authored, it is analysed to yield goals. By exploiting the goal-scenario relationship in the reverse direction, i.e. from scenario to goals, the approach pro-actively guides the requirements elicitation process. The authors introduce *requirement chunks*, which are pairs of goals and scenarios. Scenarios are composed of actions, an action being an interaction of agents. Scenarios can be normal and exceptions. The former lead to the achievement of the associated goal, while the latter lead to its failure. Goals in CREWS are encoded textually. Goal descriptions in this approach are quite complex and are expressed as a clause with the main verb and a number of parameters including quality, object, source, beneficiary, etc. In this process, goal discovery and scenario authoring are complementary steps and goals are incrementally discovered by repeating the goal-discovery, scenario-authoring cycle. Goal discovery in this method is on the linguistic analysis of goal statements. Requirement chunks can be composed using AND/OR relationship or through refinement, which relates chunks of at different levels of abstraction.

The CREWS approach is able to automatically generate a very large number of potential goal refinements by modifying the parameters of the textual representation for goals. Goals in CREWS are encoded textually. Goal descriptions in this approach are quite complex and are expressed as a clause with the main verb and a number of parameters including quality, object, source, beneficiary, etc. This is not unlike the process employed by various evolutionary computing approaches [Eib05]. Those are rooted in the ideas from the natural evolution of species and usually employ the generate-and-test methodology where a multitude of potential solutions is generated (mostly by using a recombination operation that merges several solutions from the previous "generation" to generate an offspring as well as a mutation operation) and then the fitness of each of the newly generated solutions is tested. In CREWS, the automatically generated goals are validated by scenarios. The approach can generate a very large number of alternative ways to achieve a goal by iterating through all the possible values for all the possible parameters of the goal. This large number of alternative goals must be pruned to remove meaningless goals. The way to do it is to create a scenario for it and to test whether the goal is *realistic*. Thus, in this approach, goal refinements are generated purely linguistically. The authors claim that their approach works better than ad hoc methods since in the context of RE, the larger the number of alternatives explored the better. Alternatively, variability may be introduced into the approach by varying the scenario portions of the chunks. In this case, goal parameters remain fixed, while scenarios through which the goal is achieved are changed. The approach does not suggest a way to systematically analyze alternative system configurations.

The linguistic approach by the CREWS method served as an inspiration for the work of Liaskos et al. [LLYY06] where OR decompositions of goals are characterized semantically. The work is based on Fillmore's case system [Fil68]. Fillmore states that a sentence consists of a verb and a set of noun phrases. Each noun phrase has a particular semantic relationship with the sentence. This relationship can be characterized by a semantic type. Linguistically, these semantic types correspond to *cases*. Fillmore proposes a set of such case types that would fit any language. Each such universal case type addresses a particular semantic concern that is associated with the verb. So, these cases can be viewed as defining potential semantic slots for the sentences. Then, given a particular verb, it *evokes* a set of such semantic slots, a *frame*. For instance, the verb "open" is usually associated with an objective slot (*what* is opened?) and may be associated, among other things, with the agentive slot (*who* opens?). In [LLYY06] the above ideas are applied to requirements engineering, and in particular to driving the discovery of goal variability. Here, a textual representation of a goal is taken and the frame for verb that describes some activity in that description is used to drive the alternative refinements of that goal. The elements of the frame (i.e., the cases) can be seen as *variability concerns*. Variability concerns are types of questions whose alternative answers will generate alternative refinements of the original goal. A general set of variability concerns is proposed. These are some of the concerns that are included in the set:

- **Agentive** – concerns the agents who bring about the desired state of affairs represented by the verb. This can be viewed as introducing alternative delegations of the goal.
- **Dative** – concerns the agents who benefit from the activity implied by the verb (e.g., "send a message to <who?>").
- **Objective** – is the concern of the object(s) that are affected by the goal's activity (e.g., "send <what?> to ...").
- **Process** – determines what is involved in the performance of the activity, the means, or the manner. For example, payments can be performed by debit or credit cards.
- **Temporal** – is the concern about the duration of activities that are implied by the verb.

Based on the idea of variability concerns an approach for the use of variability frames for the systematic discovery and modeling of intentional variability using goal models is proposed. This includes the construction of problem-specific variability frames to better guide the elicitation of such variability in various domains. In terms of the modeling of the concern-driven goal decompositions, multifaceted OR decompositions are proposed together with the annotations that allow for the modeler to identify whether a particular variability concern has been fully addressed in the goal model. We believe that this approach can be used together with the ideas presented in this thesis to provide systematic guidance aimed at the identification, modeling, and analysis of intentional variability in goal models. In addition to discussing intentional variability, Liaskos et al. identify "background" variability as an important factor in goal-driven requirements elicitation and analysis. This type of variability captures properties of the environment of the system. We further address the modeling of the variability in the environment of the system and its effect on requirements in Chapter 4 of this thesis.

### 2.3.2 Variability in the Solution Domain

Moving on to the solution domain, the systematic discovery and exploitation of commonality across related software systems is considered a fundamental technical requirement for achieving successful software reuse [PD90]. Thus, the software reuse community has long been interested in analyzing commonality among closely related software systems (called a *product line* or a *domain*). A method called Feature-Oriented Domain Analysis (FODA) [KCHN90] was the first to use *features* for analyzing and representing commonality and variability among applications in a domain. There, members of a software product line are characterized by their features, so variability in a product line can be represented by a feature model. The application domain is considered to be the set of current and future applications. Feature modeling [KCHN90] is a domain analysis technique that is part of an encompassing process for developing software for reuse (referred to as *Domain Engineering* [CE00]). As such, it can directly help in generating domain-oriented architectures such as product line families [KKLL99], which are families of related products. The sources of features in feature models are identified in [CE00] as stakeholders, domain experts, previously developed systems, etc.

Feature modeling is also used in varying degrees in other product line development frameworks such as FeatuRSEB [GFD98] that uses use-case analysis as the main vehicle for discovering variability in product lines, with feature models playing an auxiliary role through reflecting on the experience from the past projects. In terms of the origins of variability, many domain and product line engineering approaches list domain experts as their main sources (e.g., the DARE approach [FPF98]). Overall, the domain and product line engineering approaches concentrate on the analysis of existing and future systems and generally do not pay enough attention to similarity or variability within problems that these systems are aimed at addressing.

There are, however, approaches that use requirements analysis as a means for identification of variability and commonality. Still, these approaches are extremely solution-oriented and are not intentional. One example is the Domain Specific Software Architecture framework [Tra95] that uses sets of scenarios to produce requirements documents containing variation points. Here, requirements are not treated in a goal-oriented way. Rather, they are viewed as features of the system-to-be. Also, a special Product-Line Requirements Specification document capturing the requirements that are common to all members of a product line as well as how the requirements may vary is proposed in [Fau01]. This document can help with understanding and communicating the characteristics of a family of products and for producing specifications for individual product family members. However, no process for producing such documents is presented.

In terms of representation of variability in various solution-oriented modeling notations, Kang et al. [KCHN90] identify three ways to represent variability in a functional or behavioural model: 1) break the model into multiple models, 2) create one model that captures variability through parameterization, and 3) create a generic model that can be instantiated accordingly. The first way can usually be chosen if no other options can be applied to the problem. The second option is illustrated in [KCHN90] as well as by Gomaa [Gom00] for the use with statecharts. There, states can be OR-decomposed into alternative substates with the transition guarded by special parameters. We use a similar approach in Section 5.3.2 of this thesis to generate high-variability statecharts from goal models. Other approaches for capturing behavioural variability include the use of modal transition systems [FUB06] or message sequence charts that are augmented with variation points [ZUJ02]. Robak et al. [RFP02] add special decision elements, which are bound depending on the particular system family variation, to activity diagrams. On the other hand, in [JOZ03], workflow-level models are customized by first generating their textual representation and then automatically editing them using customization scripts. In the approach for generating high-variability executable workflow-level business process specifications from high-variability goal models (see Chapter 6) we use parameterization to support variability as well as for traceability of workflow variants to the corresponding variations in the goal models.

PuLSE [BFKL99] is another domain modeling approach that is geared towards helping a software organization determine which products are worth developing. It provides tables that represent existing, future, and possible future applications together with the distinct characteristics of these applications, their benefits and costs and whether competitors' products already have these characteristics. So, the scope of the analysis is bounded by the existing assets and potential new developments.

### 2.3.3 Variability in Software Systems

Over the last several decades, software systems increasingly exhibited the ability to vary their behaviour during their lifecycles. Svahnberg et al. [SGB05] identify several reasons for this trend. First, we see the gradual move from the variability in hardware to variability in software systems. For instance, car manufacturers often use different engine control software to create several versions of the same physical engine for different vehicle models or trims. Second, the high cost of reversing design decisions once they are taken forces development organizations to delay these decisions to the latest phase in the software system's lifecycle that is economically feasible. This trend manifests itself in the proliferation of the software product line (SPL) [CN01] use and in the increase in the customizability of software. Market pressure on software developers force them to increase the number of products they offer to better cater to various categories of (potential) customers. For most organizations, asset reuse afforded by SPLs is the only feasible option to manage these systems. Thus, they are able to exploit the commonality among their products and implement the differences among them as variability in software artefacts. Ideally, SPLs should also accommodate possible future changes, thus it is important to support the introduction of new and possibly the removal of certain existing variations in a product line. While in the previous section we surveyed some of the approaches for identifying variability in the solution domain, here we talk about how variability can be realized in software systems. We found the idea of taxonomy of variability realization techniques presented in [SGB05] to be very useful in understanding the landscape of challenges and technologies in this area.

In this thesis, we do not talk about software product lines. Rather, we discuss alternative ways to meet requirements and the corresponding alternative behaviours. These behaviours are different in some respects and similar in others and this variability needs to be preserved in implemented systems. However, we abstract from most implementation details. We believe that in general, while our goals are quite different from the aims of the SPL community, that community's ideas on implementing variability in software can be used with our approach.

Software variability is defined in [SGB05] as "the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context". Basically, software variability allows delaying design decisions until later stages of the development process. However, there are costs associated with these delayed decisions as variability must be maintained. For instance, several versions of certain system features might need to be implemented.

There are a number of steps for introducing variability into software systems. The activities are listed in [SGB05] as: 1) identify, 2) constrain, 3) implement, and 4) manage. The first step, the identification of variability, is normally done through the use of features and feature models. So the use of domain engineering approaches is appropriate at this stage. Svahnberg et al. [SGB05] advocate the use of features since they "bridge the gap between requirements and technical design decisions" and in general can be viewed as abstractions over requirements. There is no one-to-one correspondence between features and requirements (we also note this fact in Section 5.3.1 of this thesis). In the context of product lines, a product family must support variability for the features that differ from product to product. These features can be captured in feature models by appropriate variable feature types [CE00].

Once variant features have been identified, they need to be appropriately constrained. To constrain a feature is to decide how to implement the feature in a product family. Svahnberg et al. [SGB05] provide interesting ideas about the lifecycle of variant features, which has to be taken into account when choosing the appropriate realization techniques for those features. Variant features start by being *identified* (as described above). When variant features are first identified, they become *implicit* since they are still to be implemented in the product family. Their implementation at this point is considered deferred to a later phase in the development. Once a variant feature has a representation in the design or implementation of an SPL, it becomes *introduced*. This representation consists of a set of variation points, which are places in the design/implementation that provide the machinery to support the variability of this feature (thus, they are different from the variation points we discussed in Section 1.2). The variants of the feature (i.e., the particular choices within that feature) do not have to be present at this point. The introduction of these variants makes the feature *populated*. Here, software entities representing the variants have to be introduced into the design and the variation points must be capable of using these variants. When a decision about which variant to use is made, the variant feature becomes *bound*.

A variant feature is introduced into a product family as a set of VPs. The decision on when to introduce it depends on things such as the size of the affected software entities (e.g., a component, a class, or a line of code), the number of resulting variation points (the smaller the number of these, the better it is for maintainability), and the cost of maintaining the feature [SGB05]. Interestingly, there is usually a dependency between the level of abstraction of the affected software entities (e.g., a component vs. a line of code) and the number of VPs required to implement the variant feature.

When populating a variant feature with variants, there are a number of questions to consider. First, one has to think about when to populate. During some phases of the development process (not including the requirements phase) – i.e., architectural design, detailed design, implementation, compilation, linking – a VP is *open* for addition or removal of variants. During other phases it is considered *closed*. There is correspondence between the type of software entities involved in a VP and the phases when it is open. For instance, if the VP is implemented at the level of software components, then it is open during the architectural design phase and at runtime and closed in other phases. It is important to point out that when a VP is open/closed is determined from both the technical standpoint (as above: the limitations of the chosen technique) and from the point of view of when one wants to have VPs open/closed.

The population of a VP can be done either *explicitly* or *implicitly*. This depends on the way it is implemented. With the implicit population, the VP has no knowledge of the choices it possesses. Thus, the list of variants is not represented within the system. It may be maintained outside of the system. An example of this is the selection of which OS to build the system for, which is normally done outside of the system. On the other hand, with explicit population, the list of choices for the VP is available within the system. Thus, the system is aware of the alternatives, can add new alternatives to the list and has the capabilities to select the best alternative at runtime.

At some point there must be a decision made as to which variant to use. This is called the *binding* of a variant feature. Binding can be done at various phases in the system's lifecycle. At the top in terms of the level of abstraction is the phase of the *product architecture derivation*. A product line architecture contains a number of unbound VPs and binding some of these will result in a particular member of the SPL with a particular product architecture. Here, various architecture description languages such as KOALA [OLKM00] are used for describing product line architectures and for deriving products.

A variant feature can be bound at the level of compilation. Compiler directives can be used to prune the source code or introduce additional behaviour into the code through, for example, using *aspects* [KLMM97]. Variability can also be bound during linking. Exactly when linking is performed depends on the technologies/products being used throughout the development of the system and at runtime. Sometimes linking can be delayed until runtime (e.g., dynamic link libraries). Finally, variant features can be bound at runtime. In some cases, VPs can be open at runtime and new variants can be introduces (e.g., through the use of plug-in components). Also, start-up parameters or configuration files can be considered another type of runtime binding. In Chapter 6, we use goal models to generate configuration parameters corresponding to the business process alternative selected based on user preferences. This configuration is then provided to a high-variability business process at the time of its instantiation, thus customizing the process instance.

One important aspect of binding is whether it is *internal* or *external*. Internal binding requires the system to have the functionality to bind to a variant on its own, without external help. This is normally the case for runtime binding and means that the software must contain the capabilities to select among the available alternatives and to bind them. On the other hand, external binding implies that there is a person or an external tool that performs the binding. This is normally done during phases other than runtime.

When the type of population of a variant feature is combined with the type of binding, we have a number of possibilities that are important to consider when developing adaptive systems. For instance, *implicit population* together with *internal binding* implies that the system has the capabilities to select an appropriate variant, but does not have the ability to manage or extend variants. Variation points of this type are usually bound at runtime. An example of this would be an email application that supports both POP and IMAP protocols and can switch among them, but is unable to add any new protocols to its repertoire. Another possibility is *implicit population* and *external binding*. This is the most common way to manage variability for non-adaptive systems: systems using this combination do not know what alternatives exist and are not concerned with their binding. All variability is bound during various phases of the development of the system. At the other end of the spectrum is the variant with *explicit population* and *internal binding*. With this combination, the system is aware of the alternatives that exist and is capable of selecting among them. It provides the most flexibility at runtime. The selection of the approach to implement variability is done on the basis of the analysis of the individual variation points. When selecting techniques for implementation, one needs to balance complexity and flexibility. Also, too much variability (through the use of too many variation points) can lead to *architecture drift* [PW92].

Svahnberg et al. [SGB05] present a number of variability realization techniques in the form adapted from the literature on design patterns [GHJV95] identifying the intent and motivation behind them, as well as summarizing the solution, talking about the lifecycle of the techniques, describing consequences of their use and finishing up with examples. We outline some of the techniques in a very condensed form. We focus on architecture-level techniques. However, a number of source code-level approaches are also described in [SGB05].

**Architecture Reorganization**. Here, the product family architecture is reorganized to produce product-specific architectures. This is usually done by changing control flow among the components, but changes to component interfaces are also possible. Here, architecture components are controlled by configuration management tools or ADLs [MT97] such as KOALA [OLKM00]. This technique is implicit and external and there is no representation of the system architecture at runtime. In terms of the lifecycle, this technique is open during architectural design (i.e., during architecture derivation from the template) when new variants can be added. It is bound when a particular product architecture is derived from the product family architecture. This is not a technique for dynamic architectures.

**Variant Architecture Component**. The intent here is to support multiple alternative components representing the same conceptual entity. It is open during architectural design. This technique is bound when the architecture for a product is generated from a product family architecture. The binding is external.

**Optional Architecture Component**. Provides support for components that may or may not be present in particular products. This technique is open during architectural design and is bound externally. An example of this is the plugin architecture of the Mozilla browser, which provides a null-plugin that is called in the absence of a plugin to call for some embedded object.

**Binary Replacement – Linker Directive**. This technique supports alternative implementations of libraries. This may be needed when, for instance, different platforms require different versions of a library. This technique is open during linking and is bound at this stage as well. In some modern systems, this technique may be available during runtime.

**Infrastructure-Centered Architecture**. This technique enhances architecture management by converting connectors into first-class entities. This way, components are no longer directly connected to each other, but to an infrastructure (i.e., the connectors). Then, the infrastructure matches the required interface of one component with the provided interface of another one. The infrastructure can be realized using an existing component framework of can be an in-house development using, for instance, KOALA. A scripting language can be provided within the infrastructure. The variants in this approach are either explicit or implicit and the binding functionality is internal, provided by the infrastructure. Service-Oriented Architecture and its related standards/technologies (including scripting languages such as BPEL [BPEL07]) fall into this category of variability realization techniques. SOA allows components/services to concentrate on implementing business logic, while orchestration/choreography and variability management is offloaded to the appropriate middleware.

**Runtime Variant Component Specializations**. The goal of this technique is to support the existence and the selection among a number of specializations inside implemented components. Basically, this supports components with multiple implementations that can be selected at runtime based on the current circumstances. The implementation of this technique is rather low-level, using a number of design patterns such as Strategy, Abstract Factory, etc. with *inheritance* and *polymorphism* used as ways to implement variability. The technique is open during detailed design and the binding happens at runtime.

**Variant Component Implementations**. The goal here is to support a number of implementations of a component so that any of these can be chosen when appropriate. An example of when this technique is useful is the dynamic selection of either POP or IMAP protocol within an email application. The solution is to implement several alternative component implementations and make them “tangible entities in the system architecture” [SGB05]. This way the choices are explicitly represented in the system and thus the binding can be done internally. The binding is done at start-up or at runtime.

## 2.4 Adaptive Systems Design

### 2.4.1 Introduction

Research on adaptive software systems has gained a lot of momentum in the recent years. This is mostly due to the ever-increasing complexity of today’s systems and thus the dramatic growth in their management complexity. Modern software systems increasingly deal with distributed applications in changing environments. Additional complexities arise from the need of systems to support robust handling of unexpected conditions. Traditionally, software engineering aimed at handling complexity and achieving quality goals has been focused on “software development and its internal quality attributes” [ST09]. However, recently there was an increase in demand for dealing with robustness, reconfiguration, etc. at runtime. This is mostly due to the increase in complexity, heterogeneity, the frequency of changes of the systems’ environments, etc. In this context, troubleshooting, reconfiguration, and maintenance require expensive and lengthy procedures involving human operators. This is due to the fact that many software systems have a simplistic open-loop structure. Adaptive systems’ goal is to offload that management complexity from human operators into the software itself. Adaptive systems are *closed-loop* systems that employ feedback loops aimed at adjusting the behaviour of systems at runtime based on the changing circumstances. Such closed-loop systems need to *monitor* themselves and their environments, *detect* significant changes that require adaptations to their behaviours, *decide* how to react to those changes (e.g., by doing *diagnosis*, planning, etc.), and *act* to execute the decisions.

There are two major types of changes requiring system adjustments that are identified in adaptive systems research: changes that originate within the system itself (sometimes called the *internal context*) and changes originating in the system's environment (*external context*). The latter can include, for example, changing user preferences, etc. Thus, adaptive systems aim at adjusting their artefacts or attributes in response to changes in either internal or external context. In other words, adaptive systems' mission is to fulfil their requirements at runtime in response to changes [ST09].

There are several terms that are used in the literature to describe essentially the same class of systems that are self-managing in a number of aspects. These are sometimes called *autonomic computing* systems [KC03], self-managing systems, or self-adaptive systems. Most researchers use these terms interchangeably [HM08]. Still, some find differences between the terms. For example, Salehie and Tahvildari [ST09] state that the self-adaptive software domain is more limited, while autonomic computing "has emerged in a broader context", which means that self-adaptive software "has less coverage and falls under the umbrella of autonomic computing."

There are a number of definitions of self-adaptive software. One typical definition is provided by the DARPA Broad Agency Announcement. It is quite operational and states that "self-adaptive software evaluates its own behaviour and changes behaviour when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible" [Lad97]. Similarly, Cheng et al. state that self-adaptive systems are "systems that are able to adjust their behaviour in response to their perception of the environment and the system itself" [CLGI09].

The main idea in self-adaptive/autonomic/self-managing software is that its lifecycle is not stopped after it has been developed, delivered, and setup. Rather, it should be continued at runtime through monitoring, evaluation, and response to changes. Many application areas of computer science and software engineering have focused on systems with self-adaptive behaviour. Examples include such diverse areas as adaptable user interfaces, dependable computing, embedded systems, multiagent systems, ubiquitous computing, service-oriented architectures, etc. Despite the interest in such systems, “little endeavour has been made to establish suitable software engineering approaches for the provision of self-adaptation” [CLGI09].

According to Huebscher and McCann [HM08], one of the first important self-management projects was a military project started by DARPA in 1997. Then, one of the more prominent initiatives in the area of self-adaptive and self-managing systems was Autonomic Computing ([Hor01], [KC03]) proposed by IBM in 2001. The term “autonomic” comes from biology. In humans, autonomic nervous system takes care of unconscious reflexes such as the digestive function, the heart rate, etc. With these activities offloaded to the autonomic nervous system, humans do not have to continuously and consciously adapt their bodies to their own needs and to the needs of the environment. Thus, autonomic systems try to mimic the autonomic nervous system by taking on self-management activities (i.e., the regular maintenance and optimization tasks) and thereby freeing their human operators from the need to manually manage a lot of the aspects of these systems. The main focus of IBM’s research on autonomic computing was systems management.

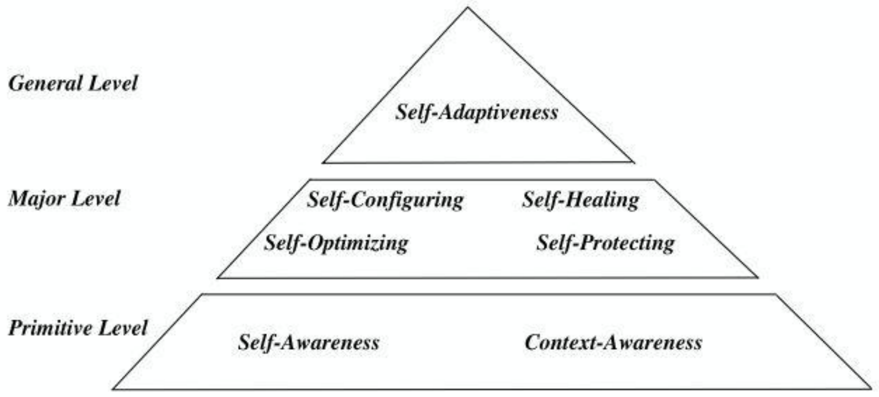


Figure 2-1. Hierarchy of Self-* properties [ST09].

### 2.4.2 The Self-* Properties

Self-* or adaptivity properties are properties that must or may be possessed by self-adaptive systems. Depending on the application and the desired characteristics of the adaptive systems, they might have different subsets of these properties. The properties self-configuration, self-protection, self-optimization, and self-healing were proposed by IBM [KC03] and were positioned as the main properties of autonomic systems. Since then researchers came up with a number of other self-* properties. They can be organized into a 3-level hierarchy [ST09] (see Figure 2-1).

The top level in the hierarchy consists of the global properties that adaptive software systems strive to have. *Self-adaptiveness* can be seen as a subset of these properties encompassing *self-managing, self-control*, *self-governing*, *self-maintenance*, and *self-evaluation*. A different subset of the general self-* properties is *self-organizing.* It emphasizes decentralization and emergent behaviour. A self-organizing system typically consists of a number of communicating elements that do not necessarily have a full view of the system architecture. The global behaviour of the system *emerges* from these local interactions among system components. It is easy to see that systems that have the self-adaptiveness property are typically organized top-down, with more central control. On the other hand, self-organizing systems are mostly organized in a bottom-up fashion.

The major level of the taxonomy consists of the four self-* properties proposed by IBM. These are designed to support the self-* properties that exist at the general level of the taxonomy. Here are the four properties at the major level. *Self-configuration* is the ability of a system to reconfigure automatically in response to changes by installing, updating, integrating, etc. software entities. *Self-optimization* is the capability of managing the system's performance (or resources) in order to satisfy the requirements of various users. *Self-protection* is the capability of detecting security problems in the system (such as hacker attacks) and recovering from them. *Self-healing* (which is linked to self-diagnosing and self-repairing) is the capability of "discovering, diagnosing, and reacting to disruptions" [ST09]. Self-diagnosing focuses on diagnosing errors, faults, and failures, while self-repair concentrates on recovering from them.

It is easy to see that in order to implement most of the higher-level adaptive properties, self-adaptive systems need to be aware of their environments as well as of themselves. Thus, at the primitive level, we have the properties *self-awareness,* which means that the system is aware of its own behaviour and state, and *context-awareness*, which means that the system is aware of its operational environment. Both of these properties rely on monitoring.

It is often noted that the self-* properties have a close relationship with certain software quality factors [ST09]. For instance, self-configuration impacts a number of factors such as portability, maintainability, etc. Similarly, self-healing affects reliability, resilience, availability, etc.

### 2.4.3 Modeling Adaptive Systems

There is no canonical way of making a system adaptive. The precise techniques to be used in such a system will depend on many aspects including the purpose of the system, the needs of the user, the characteristics of the environment, the availability of resources during development, and so on. Thus, there needs to be an effort applied to understanding the specifics of the problem. Modeling of various aspects of the adaptive system, its users, and its environment is therefore needed. Andersson et al. [ALMW09] recognize the lack of consensus from both researchers and practitioners on the points of variation among various adaptive systems. They identify a number of such points of variation and call them *modeling dimensions* for self-adaptive systems. They suggest four groups of dimensions that deal with 1) self-adaptivity aspects of system goals, 2) causes of adaptation, 3) the mechanisms to achieve adaptability, and 4) the effects of adaptability on a system. Within each of the four groups a number of dimensions are identified. The goal of this research was to come up with a conceptual model of adaptation that is independent of the technologies and tools used for its implementation. We describe some of these dimensions and also mention whether and how they are supported in the approach proposed in this thesis.

As an example, let us look at the modeling dimensions related to system goals. In [ALMW09], the following are some of the dimensions recognized in this group:

- *Evolution*: This dimension identifies whether the goals can change during the lifetime of the system. The values are *static* to *dynamic*. Evaluating our approach from this point of view, we can see that while the basic goal modeling notation supports static goals only, the addition of the context framework (Chapter 4) adds the capability of modeling goal change depending on the different environmental (and internal) conditions.

- *Flexibility*: This dimension identifies if the goals are flexible the way they are expressed. It captures the level of uncertainty in the goal specification. The values are: *rigid, constrained*, and *unconstrained*. Rigid goals are prescriptive, while unconstrained ones provide flexibility in dealing with uncertainty. In our approach, we are dealing with rigid goals only.

- *Duration*: This dimension deals with the validity of goals throughout the lifetime of the adaptive system. The values range from *temporary* to *persistent*. Goal model annotations that are utilized in our approach (see Section 3.1) can be used to attach conditions to goals, thus making them temporary. Similarly, the context framework of Chapter 4 can be used to restrict the persistence of goals.

- *Dependency*: This dimension captures whether system goals are independent of each other. For example, goals can be conflicting. Goal models used in our approach are able to specify how goals are related to each other.

In terms of the causes of adaptation (or *change*), [ALMW09] identifies the modeling dimensions such as:

- *Source*: Either *external* or *internal*; if internal, then *application*, *middleware*, or *infrastructure* values are possible. In our approach, sources of adaptation are not explicitly modeled. However, if the context framework of Chapter 4 is used, then a change in context can be regarded as a source of adaptation. Furthermore, depending on the definition of the context this change can usually be categorized as being internal or external with respect to the system.

- *Anticipation*: This dimension captures whether the changes can be predicted in advance. The values here are: *foreseen* (taken care of), *foreseeable* (planned for), and *unforeseen*. Depending on the desired change prediction level, different techniques have to be used for constructing adaptive systems. Our approach explicitly represents variability in the problem and in the solution domain, thus covering the foreseen changes. A possible integration of the approach proposed in this thesis with agent technology [LL10] can support foreseeable changes with the introduction of planning capabilities into the system.

In Chapter 8 of this thesis, we propose an approach for systematic elicitation and modeling of monitoring requirements for adaptive systems. Moreover, we use contexts to model hierarchies of possible *adaptation situations* – conditions in the environment and/or within the system that warrant adaptation along with explicit representation of alternative compensation mechanisms available in each of the cases. All of this allows the modeling of monitoring, analysis, and adaptation explicitly and at the intentional level.

Andersson et al. [ALMW09] further identify modeling dimensions capturing the adaptation mechanisms. For example, adaptation *type* is described as either *parametric* or *structural* with the former involving the modification of the system parameters while the latter involving more substantial changes to the system (e.g., replacement of components). This is similar to what is described in [ST09] as *weak* and *strong* adaptation actions respectively. In our proposal, we abstract from the implementation details. Adaptations from the point of view of goal models involve the selection of appropriate ways to achieve goals. Whether these are realized through parameterization or component replacement is not captured.

*Adaptation autonomy* is another modeling dimension related to the mechanisms of adaptation. It has two values: *autonomous* and *assisted*. In our approach, the explicitly specified alternatives provide for autonomous adaptation. However, in the case of unforeseen changes, human assistance is likely needed. Goal models are high-level and capture intentional aspect of systems. Thus, they are quite accessible even for non-IT people and can be used to keep humans in the loop.

Further dimensions in the category of adaptation mechanisms include: organization (*centralized* vs. *decentralized*), scope, duration, timeliness, and triggering. Most of these deal with the level of detail that that we do not capture using goal models.

Finally, there is a set of modeling dimensions that have to do with the impact of adaptation on the system. For example, one dimension captures predictability of the effects of an adaptation on the system (*non-deterministic* vs. *deterministic*). In our approach, one of the main features is that adaptation possibilities are modeled and analyzed explicitly. This supports deterministic adaptation. Other dimensions in this category include resilience, overhead, and criticality.

### 2.4.4 Requirements Elicitation for Self-Adaptive Systems

Requirements elicitation and modeling for adaptive systems presents a challenge since in addition to the usual requirements engineering activities aimed at producing functional and non-functional requirements for the system, we are presented with the task of doing additional RE work aimed at capturing the requirements for the adaptive infrastructure including monitoring, analysis/diagnosis, planning, compensation/recovery, etc. Not all approaches devote enough attention to the problem of requirements elicitation for adaptive systems. For example, the seminal paper on autonomic computing [KC03] glosses over this problem almost completely.

Berry, Cheng and Zhang discuss the fundamental activities in RE for dynamic adaptive systems (DAS) [BCZ05]. To be truly dynamic and adaptive, such systems need to do RE at runtime. In other words, since RE is essentially about inputs and responses, a DAS $S$ has to determine the kinds of inputs it may be faced with and its responses to the inputs. Further, an adaptable system $S_{AR}$ (adapt-ready) with an input space $D$ (its domain) is defined. $S_{AR}$ is seen as having many target programs $S_i$ – the programs that exhibit one of the behaviours that $S_{AR}$ can adopt after adapting, each with its own domain $D_i$. The set of all the target programs $S_i$ is $S$. The authors then postulate that there exist four levels of RE for and in $S_{AR}$. They are identified in order of their meta-level with the level $j$+1 making decisions about the subject matter of the level $j$. Here are the levels:

1. Level 1 RE is done by humans for all the target programs in $S$ to determine the input spaces $D_i$ for each $S_i$ and to determine the target program's reaction to each input in $D_i$.

2. Level 2 RE is done by the $S_{AR}$ system during its own execution in order to decide based on the input whether it needs to adapt and to determine which of the behaviours ($S_i$) to select.

3. Level 3 RE is done by humans in order to determine $S_{AR}$'s adaptation elements (e.g., monitoring infrastructure, decision-making and diagnostic facilities), which allow $S_{AR}$ to do the adaptation expressed at Level 2 RE. In essence, this is the work done for a particular adaptive application to select and configure its adaptation infrastructure.

4. Level 4 RE is done by humans to discover adaptation mechanisms in general. This can be viewed as the RE research for adaptive systems.

In general, Level 3 RE will happen before Level 2 since it determines the capabilities available at Level 2. Basically, the main activities for requirements engineers in Levels 1, 2, and 3 are the identification of the set of target programs, the methods for choosing among them, and the monitoring, analysis, and adaptation techniques. It is also possible that some of these activities will have to be performed after the system has been deployed and running. The reason is that when the system is presented with an unanticipated input (not within the input space *D*) additional Level 1 activities will have to be performed to identify the appropriate behaviour to handle the new input (the new target program) as well as new Level 3 activities will need to be done to revise some of the adaptation mechanisms of the DAS.

In [ST09], a set of six questions is used to help with elicitation of adaptation requirements. The questions are:

- *Where*: Where in the system is the need for change? Which artefacts need to be changed?
- *When*: When do we need to apply the change? How often do we need to change the system? Is the system going to change reactively or does it need to act proactively?
- *What*: What attributes or artefacts do we need to change using adaptation actions? Available alternatives for adaptation actions need to be identified.
- *Why*: Here, we are dealing with the rationale/motivation for adaptive behaviour. These need to be rooted in the systems' objectives.
- *Who*: Here, we are dealing with the level of automation in a self-adaptive system (i.e., the degree of human involvement).
- *How*: How can the selected artefacts be changed with adaptation actions?

What is more is that these questions need to be answered during the development of the system to build the software and to setup the appropriate adaptation infrastructure and then at runtime the system needs to actually adapt based on the answers to those questions. The answers at runtime will be highly dependent on the type of approach for adaptation chosen for the system.

### 2.4.5 Implementing Adaptive Software

There are many techniques used for implementing adaptive software. A number of research efforts were devoted to characterizing the spectre of such techniques. Oreizy et al. [OGTH99] discuss them, first concentrating on static approaches and then moving on to dynamic ones. A taxonomy of self-adaptation is introduced in [ST09] (see Figure 2-2). The authors also attempted to relate the facets of the taxonomy with the set of requirements question discussed in Section 2.4.4.

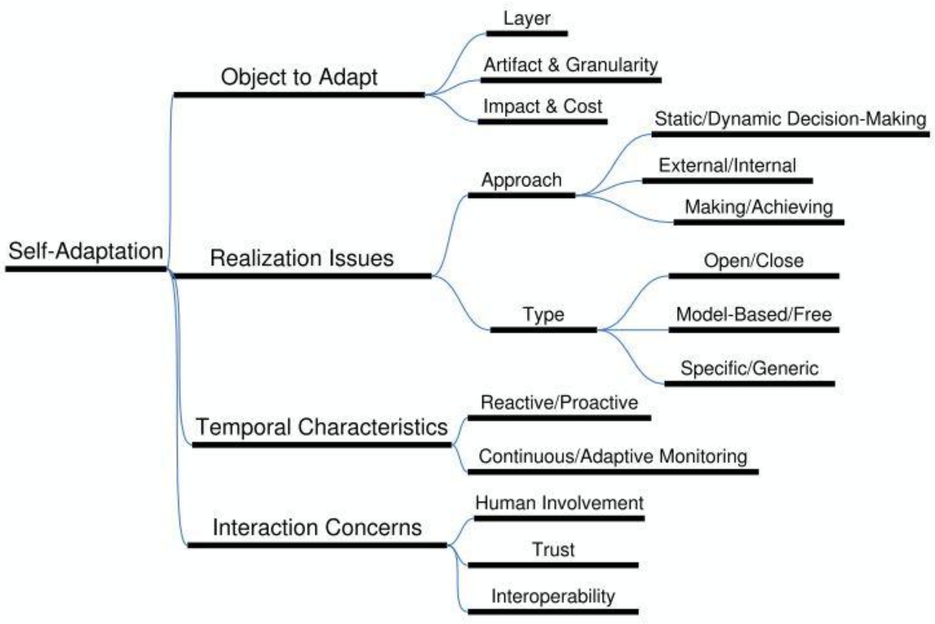


Figure 2-2. A taxonomy of self-adaptation [ST09].

We now briefly discuss the facets of the taxonomy. *Objects to Adapt* is a facet related to the *what* and *where* aspects of change in the adaptive system. It is further decomposed into the following sub-facets:

- *Layer*: Which layer of the system needs to be changed? The idea is that various adaptation actions can be applied to appropriate layers within the system. For instance, [MSKC04] defines two layers: application and middleware. One difference between the two layers is the visibility of the changes in these layers to the users of the system. Goal models used in our approach make it possible to explicitly represent available behaviour alternatives and visualize the system in terms of its current configuration as well as the location of failures and their effect on system goals. Thus, the approach is well-suited to handle adaptations at the application layer where trust, predictability, and transparency are of paramount importance. However, nothing precludes it to be used at the middleware layer.

- *Artefact and Granularity*: *What* particular artefact, attribute, etc. needs to change? Depending on the building blocks of the application (e.g., components, services, etc.), the granularity of adaptation can be different. In our approach we generally abstract from these details. What is important for us is the intentional level: the change in goals and ways to attain them.

- *Impact and Cost*: This facet describes what impact an adaptation has on the system and its cost in terms of time, resources, etc. Based on these criteria, adaptation actions can be characterized as *weak* or *strong*. Weak adaptation can be viewed as modifying parameters and performing actions with low cost and limited effects (e.g., static load balancing or data compression). Example of weak adaptation actions are caching [OGTH99], changing data quality or the type of data (video vs. image) [MG05], tuning [KLSP01], load balancing, etc. On the other hand, strong adaptation deals with expensive actions with broader effects. An example of such action would be component addition, removal, or replacement [MSKC04] or changing system architecture [KM90].

The facet *Realization Issues* of the taxonomy describes *how* the adaptation is to be applied. It is further categorized into *Approach* and *Type*. The approach to self-adaptation is described by the following sub-facets:

- *Static/Dynamic Decision-Making*: This deals with how the decision process is implemented in self-adaptive software. In the static case, it is hardcoded into the system in some way (e.g., as a decision tree). Changes to the decision mechanism require recompilation of the system. On the other hand, dynamic decision-making features externally defined policies [KW04], rules [LPH04], or quality of service parameters. This way these definitions can be updated at runtime, thus effectively changing the behaviour of the system.

- *Internal/External Adaptation*: Adaptive behaviour can be viewed as either being internal or external with respect to the system. These two choices are illustrated in Figure 2-3, which is taken from [ST09]. Basically, the *internal* approach mixes application and adaptation logic. Usually, approaches using this method of implementing adaptive behaviour rely on the features of programming languages – conditional expressions, parameterization, exceptions, etc. (e.g., [OGTH99], [FHSE06]). Here, the adaptive infrastructure including sensors and effectors is integrated into the application code, which hinders scalability and makes testing and maintenance harder. System evolution is also problematic since the adaptation logic is intertwined with the application logic.

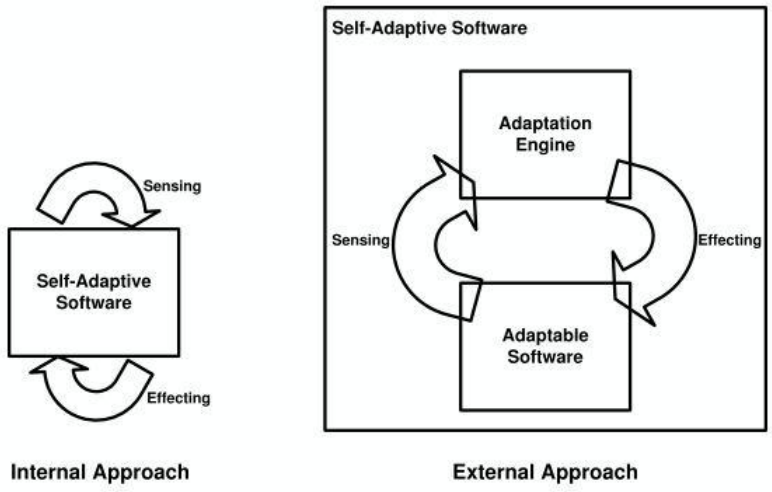


Figure 2-3. Internal vs. External adaptation [ST09].

On the other hand, *external* adaptation approaches use external adaptation engines. This is where the adaptation processes such as monitoring and diagnosis are located. Here, the self-adaptive system consists of an adaptation engine and an adaptable software, which is controlled by the engine. The external adaptation engine is where the adaptation logic is implemented, frequently with the help of middleware [FHSE06], a policy engine [BSPM02], etc. Quite often, a large adaptive system requires multiple self-adaptive elements (e.g., [KC03]) composed together in some kind of architecture, possibly a hierarchy. Adaptation managers are usually implemented as variations of feedback controllers. The significant advantage here is the ability to reuse the adaptation infrastructure as well as to come up with reusable methods for tuning this infrastructure for the needs of particular applications. The approach presented in Chapter 8 of this thesis can be viewed as a starting point for such customization. We discuss feedback loops in more detail in the next section.

- *Making/Achieving Adaptation*: Horn [Hor01] identifies several ways in which self-adaptivity can be introduced into systems. First, it can be engineered into the system throughout the development (*making*). This approach takes the software engineering view on adaptive systems. Second, self-adaptivity can be achieved through adaptive learning (*achieving*), which is an artificial intelligence view. Combinations of these approaches can be beneficial. In [LL10] we argue that feedback-loop-based SE methods for engineering self-adaptive systems can and should be augmented with agent and/or AI technologies in particular situations such as when dealing with incompletely specified domains, unknown system goals, etc. This remains part of the future work.

Another sub-facet of the *Realization Issues* facet is the *Type* of adaptation, which is in turn refined into:

- *Close/Open Adaptation*: *Close* adaptation means that an application only has a predefined number of adaptation actions and behaviour alternatives and no new actions or behaviours can be introduced at runtime. On the other hand, in *open* adaptation, adaptive software systems can be extended and new alternatives can be added (e.g., through *aspects*). This facet is closely linked with the way variation points are implemented in software (see Section 2.3.3). In our work presented in this thesis, we concentrate on close adaptation with available alternatives explicitly specified. While being a limitation, this improves transparency and predictability of self-adaptive systems.

- *Model-Based/Free Adaptation*: Model-free adaptation does not use any predefined model for the system and its environment. An example of this approach would be model-free reinforcement learning. In model-based adaptation, adaptation mechanisms use models of the system and its environment developed using various modeling techniques. For instance, a queuing model aimed at self-optimization [LWZ05], architectural models for self-healing [GS02], etc. We propose the use of goal models at runtime to facilitate adaptation.

- *Specific/Generic Adaptation*: Certain existing approaches for adaptive systems design are specifically targeting particular application domain (e.g., databases). Generic, configurable approaches are also available [LPH04]. We aim at developing a generic approach for requirements-driven design of adaptive systems.

The *Temporal Characteristics* facet of the taxonomy deals with when system components can be and have to be changed. In [ST09], the following sub-facets are identified:

- *Reactive/Proactive Adaptation*: This property deals with the way adaptive systems anticipate change. In case of reactive systems, they respond when the change (either within themselves or in their environments) has already occurred. On the other hand, proactive systems are capable of anticipating change and thus changing their behaviour before the change occurs.

- *Continuous/Adaptive Monitoring*: This sub-facet describes whether the monitoring infrastructure constantly collects the set amount of data or monitors the minimum of data while increasing the amount of data collected once an anomaly is found. The approach of Wang, et al. [WMYM09] is an example of a method that supports adaptive monitoring.

The last facet of the taxonomy is *Interaction Concerns*. It describes the features of self-adaptive systems that deal with interactions with humans and other systems. It is in turn decomposed into the following sub-facets:

- *Human Involvement*: This facet is related to the question of *who* discussed in Section 2.4.4. There are two aspects of human involvement in adaptive systems. First, we can evaluate how automated the system is. Second, the system can be analyzed in terms of how well it interacts with users and administrators. While more automation is generally good for adaptive systems, human involvement to both express their expectations, goals, and policies and to assess the current state of the system is important. In our approach, we use high-level goal models as both the way to capture stakeholder goals (as well as the multitude of ways they can be achieved) and to possibly picture what is happening in the system. The latter can involve the visualization of the currently selected alternative, the failures and their impact on higher-level goals of the system, etc.

- *Trust*: Trust has a number of definitions and can be seen as a measure of how much users rely on self-adaptive systems to achieve their goals. This view is related to assurance and dependability. One important aspect of trust is that it is not necessarily related to the quality of services provided by the system. Huebscher and McCann [HM08] discuss how trust can be built and reinforced through keeping the users of autonomic systems updated with critical information about the system status as well as by exposing the adaptation processes. This relates to the issue of predictability, which we consider extremely important, especially for self-adaptive applications (as opposed to middleware). In our approach, alternative behaviours are explicitly modeled (derived from requirements-level variability) and analyzed. Moreover, in Chapter 8, we propose explicit modeling and refinement of adaptation goals, which also increases the visibility of the adaptive processes to the users and thus helps with fostering their trust in the system.

- *Interoperability Support*: This facet is about supporting the integration and coordination of the components of self-adaptive systems. This area presents a lot of challenges from the point of view of achieving global system goals while coordinating adaptations aimed at achieving various quality requirements. As self-adaptive systems become more complex, the number of components they contain will increase. With many decentralized components (and possibly some non-determinism as well), we may begin seeing emergent features. There has been some interest in researching these emergent behaviours and attempts to have *engineered emergence*, which is defined as "purposeful design of interaction protocols so that a predictable, desired outcome or set of outcomes are achieved at a higher level" [Ant06].

### 2.4.6 The Feedback Loop

Self-adaptive systems are generally viewed as closed-loop systems that have some kind of feedback mechanism allowing them to adjust their performance to meet their objectives. In recent years, software engineering researchers started looking for new approaches to system design since traditionally the discipline of software engineering mostly focused on static system architectures and neglected their dynamic aspects. One of the few exceptions in the SE literature was the paper by Magee and Kramer on dynamic structure in software architecture [MK96], which became the basis for a number of research projects (e.g., [OGTH99]). However, while these were feedback systems, the actual feedback loops were hidden or abstracted [BMGG09]. Feedback loops were recognized as important factors in software process management and improvement as well as in software evolution. For instance, feedback loops are present at every stage in the Royce's waterfall model [Roy70]. With the growing emphasis on self-adaptive systems "it is imperative to develop theories, methods, and tools around feedback loops" [BMGG09]. The use of feedback loops allows for the separation of concerns: the main application and the adaptation infrastructure can be developed independently. Moreover, the generic adaptation infrastructure can be reused.

#### 2.4.6.1 Feedback Loop in Control Engineering

Control engineering has long emphasized feedback loops and treated them as first-class entities [HDPT04]. The main idea of feedback control is to use measurements of a system's outputs to achieve externally specified goals [HDPT04]. The goal of a feedback loop is usually to maintain properties of the *system's output* at or close to its *reference input* (see Figure 2-4). The *measured output* of the system is evaluated against the reference input and the *control error* (or *measured error*) is produced. Based on the control error, the controller decides how to adjust the system's *control input* (or *system input*) – parameters that affect the system – to bring its output to the desired value. To do that, the controller needs to possess a model of the system. In addition, a *disturbance* may influence the way control input affects the output. *Sensor noise* may be present as well. This view of feedback loops does not concentrate on the activities within the controller itself. That is the emphasis of another model of a feedback loop, often called the *autonomic control loop* [DDFG06]. We present it later in this section. It focuses on the activities that realize feedback: monitoring, analysis/diagnosis, execution.

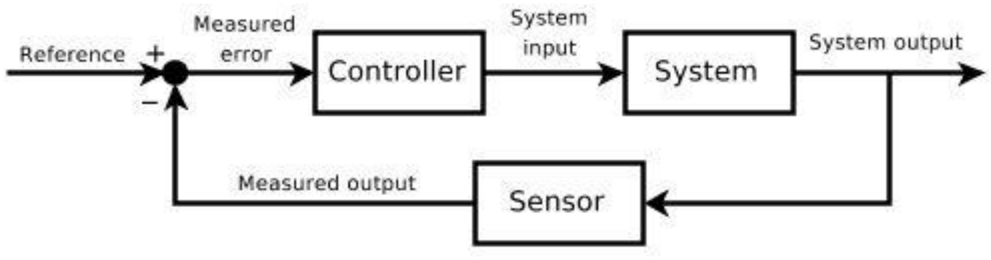


Figure 2-4. Basic feedback loop.

The common control objectives of feedback loops are regulatory control (making sure that the output is equal or near the reference input), disturbance rejection (ensuring that disturbances do not significantly affect the output), constrained optimization (obtaining the "best" value for the measured output) [HDPT04]. Control theory is concerned with developing feedback control systems with properties such as *stability* (bounded input produces bounded output), *accuracy* (the output converges to the reference input), etc. While most of these guidelines are best suited for physical systems, many of the ideas can be used for feedback control of software systems.

Adaptive control in control theory involves modifying the model of the controller in order to be able to cope with changes of the controller process. The way it is normally done is through the use of a secondary control loop installed on top of the main controller. This second control loop makes changes to the model of the controller and usually operates at a much slower pace than the main feedback control loop. Adaptive control is done in two ways: either by using a reference model of the system or by dynamically building a reference model by observing the controlled process.

### 2.4.6.2 Feedback Loops in Software Engineering

While self-adaptation in software systems is implemented in a variety of ways, what is common among these systems is that design decisions are moved towards runtime to control dynamic software behaviour and that systems must reason about their environments and themselves [BMGG09]. Feedback loops provide generic mechanism for self-adaptation.

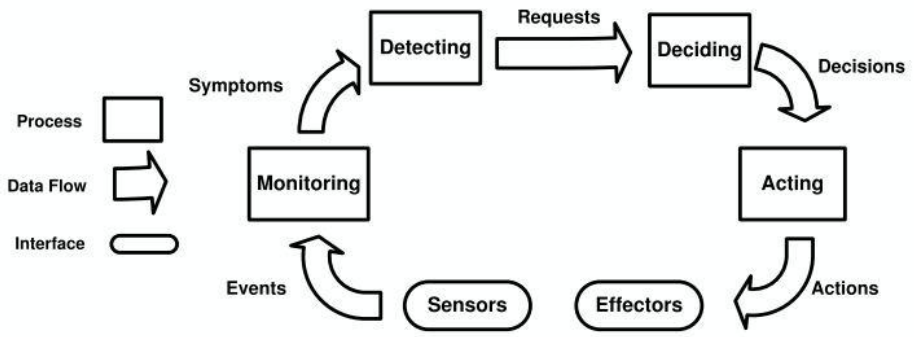


Figure 2-5. Adaptation loop [ST09].

A feedback loop typically involves a number of activities or processes as well as sensors and effectors that together realize feedback. For instance, the adaptation loop presented in [ST09]

(Figure 2-5) shows the monitoring, detecting, deciding, and acting processes. Kephart and Chess [KC03] present the autonomic control loop (or MAPE-K loop) consisting of monitoring, analysis, planning, and execution activities supported by a knowledge base. In that approach, a self-adaptive entity, an *Autonomic Element*, has two parts, the *Autonomic Manager* (AM) that implements the feedback loop and the *Managed Element*, which is controlled by the AM. Still, another version of the loop is presented in [DDFG06] and identifies the processes within the loops as collect, analyze, decide, and act (Figure 2-6). Oreizy et al. refer to this loop as the adaptation management [OGTH99]. Many adaptive systems researchers (e.g., [BMGG09], [HM08]) point out that the generic model of a feedback loop is a refinement of the sense-act-plan idea from artificial intelligence that was originally proposed to control mobile autonomous robots.

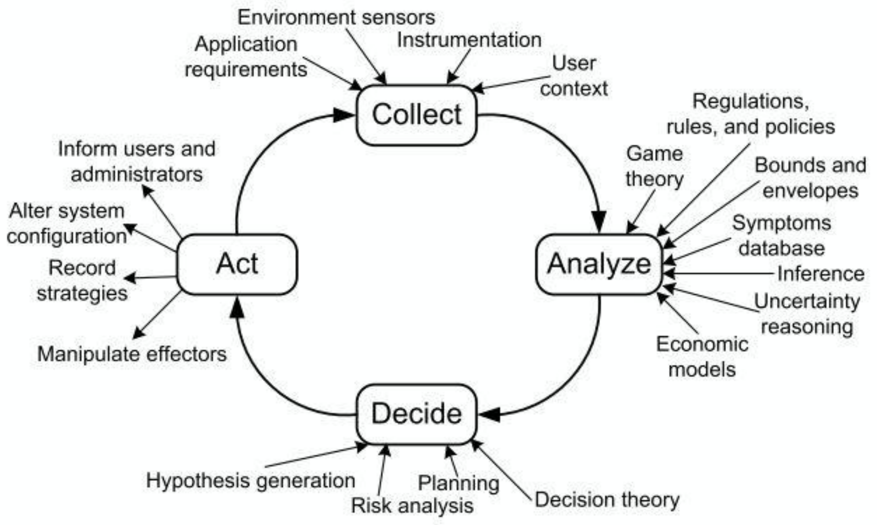


Figure 2-6. Autonomic control loop [DDFG06].

The adaptation processes that are part of the feedback loop exist at the operating phase. The feedback cycle starts with the monitoring process that is responsible for collecting and correlating relevant data from sensors and other resources that reflect the current state of the system. The aim is to capture appropriate data that will allow recognizing failures or suboptimal performance in the system. The data then needs to be correlated and analyzed. In order to be able to capture data the system needs sensors. The simplest technique for gathering data from software is logging. A number of tools exist to help in this area (such as IBM Log Trace Analyzer [IBM05]). Also, there were a number of efforts to standardize event representations across applications to simplify monitoring in heterogeneous environments (e.g., Common Base Events [IBM05]).

Active research in monitoring focused on many aspects relevant to this area. For instance, Zhang and Figueiredo [ZF06] looked into the problem of deciding which subsets of the many performance metrics in a dynamic environment to use in the most cost-effective way. They observed that a small subset of metrics provided 90% of their application classification accuracy. There are also approaches that provide dynamic monitoring capabilities. For example, QMON [ACMS06] is an adaptive monitor that changes its monitoring frequency and the volume of data it collects as to minimize the monitoring overhead while maximizing the utility of collected data. Another approach uses pulse monitoring (reflex signal) [HS06], which is an extension of heartbeat monitoring process.

The analysis process is responsible for analyzing the symptoms identified in the monitored data (as well as possibly based on the history of the system) to determine whether a response (a change in the system) is required. The approaches for analysis of the captured data include the use of models, rules, and theories. Various diagnosis methods can be used here. For example, Wang et al. provide a goal-model-based method for diagnosing systems [WMYM09] based on log data.

Then, a decision needs to be made to determine how to adapt a system in order to reach a desirable state. Different approaches for decision-making are used depending on how controller objectives are specified. Huebscher and McCann [HM08] provide an extensive discussion about how objectives for feedback loops are stated (since they talk about autonomic computing, they refer to feedback loops as autonomic managers, AMs). AMs can be configured by human administrators using high-level goals. These goals are sometimes expressed as event-condition-action (ECA) policies, goal policies, or utility function policies [KW04]. Policies are written by human administrators and determine the actions to take upon the occurrence of an event and provided that certain conditions hold. The biggest strength of policies is their relative simplicity. However, for complex systems they can become tedious to write. The difficulty with the ECA-based policies is that with multiple policies specified conflicts may occur that are hard to detect and resolve. For example, if several policies, one requiring the assignment of additional resources to the application A and another to B, fire, but the amount of available resources is not enough to fulfill both requests, a conflict arises. Here, additional conflict resolution mechanisms are needed. Thus, sizable effort has been devoted to conflict resolution techniques (e.g., [LS99]). An additional complication, however, is that conflicts may manifest themselves only at runtime.

Goal policies are higher-level in that they capture criteria that characterize desirable states, but do not specify how such states are to be found. Therefore, they require planning capabilities to be available as part of the adaptation infrastructure. A planning-based adaptation engine may need to support continuous planning through contingency planning or replanning [RN95]. Based on the sensed information contingency planning generates a conditional plan with multiple alternative paths. Replanning is used to generate an alternative plan when the original fails. Planning-based adaptation was used for self-healing by Arshad et al. [AHW03].

One difficulty with planning in the AI sense is that when a desirable state cannot be reached, the system does not know which among the undesirable states is the best. Utility functions help with this problem by assigning a quantity of desirability to each state. In turn, the problem with utility functions is the difficulty in defining them. Still, efforts are currently being devoted to approaches for using utility functions in adaptive systems (e.g., [WTKD04]). As pointed out in [ST09], decision theory in both classical and quantitative forms can contribute to the implementation of the planning/decision process in the adaptation loop in several ways. The classical version is applicable to cases where decisions involve maximizing certain utility function deterministically. Uncertainty is addressed by the quantitative version. There are a number of approaches using utility or decision theory for supporting self-* properties ([BDKW03], [RL05]). Various AI techniques such as learning algorithms and fuzzy logic can be useful in self-adaptive systems as well. Multiagent systems (MAS) have been heavily used to develop self-organizing software since they support various coordination mechanisms and distributed optimization. As already mentioned, when dealing with many independent self-optimizing components, the issues of emergent behaviour become important. MAS research provides techniques for analysis and verification of these systems. The downside is that these techniques require heavy formalization.

The planning/decision-making phase of the adaptation process requires the controller to have the appropriate knowledge about the managed system (unless the adaptations are implemented as rather simplistic ECA rules or the like). The importance of this knowledge is highlighted by the fact that the autonomic feedback loop proposed in [KC03] is called the MAPE-K loop: monitor-analyze-plan-execute over a knowledge base. This knowledge usually comes in the form of models, which could be formal or not. Pavlovic [Pav00] notes that coupling software systems with their formal models and specifications helps with monitoring correctness and many other metrics with respect to the specifications and self-* properties. Formal models can also be used for validation and verification of self-adaptive software to make sure systems function correctly and to gain understanding of their behaviour [LRS03].

Models of software architecture and architectural description languages have been heavily used by adaptive systems researchers and show that they can be helpful at runtime. A survey of a number of ADLs based on process algebras, graphs, etc. applicable to the specification of dynamic software architectures is provided in [BCDW04]. A notable architecture-based approach to self-adaptation is presented in [GCHS04] where an ADL is used to represent the architecture of adaptive systems and to detect violations from the specified constraints. In [GSC03], feedback control is used for self-repair. Usually, an architectural model does not describe a precise configuration of components and connectors that the system must conform to. Instead, it provides constraints and properties on the component/connectors that allow for determining whether the system violates the model and thus needs adaptation [HM08]. The Acme adaptation framework [SG02] is a software architecture that uses an architectural model to monitor the system and detect the need for adaptation. The architectural model contains constraints for detecting such adaptation needs. An imperative language is employed to specify repair strategies. Unlike policies, where adaptation is explicitly described in the rules, here the need for adaptation implicitly arises when constraints are violated in the running system.

As an alternative to utilizing architectural models for adaptation, [HM08] describes the use of a process coordination language Little-JIL [WCLM00] that is intended for planning tasks and coordinating components needed for the execution of specific subtasks in the plan. The language allows for specifying alternatives for completing tasks in case components responsible for their subtasks have failed or are unavailable.

In terms of other approaches for implementing adaptive systems, Salehie and Tahvildari [ST09] point to *component-based software engineering* approaches that provide a useful paradigm for implementing adaptive systems. Aspect-oriented programming [KLMM97] can be used to encapsulate adaptation concerns through dynamic runtime adaptation and to implement adaptation actions at a level that is lower than that of components. Service-Oriented Architecture can help with implementing self-adaptive systems by supporting the composition, orchestration, and choreography of loosely-coupled services.

# Chapter 3
# The Goal Modeling Notation

## 3.1 Goal Models

In this section, we present the goal modeling notation that is used throughout this thesis. Our modeling notation can be seen as a restricted version of *i** [Yu97]. While the fundamental concepts of the *i** notation include actors and intentional dependencies, our goal models are simpler in the sense that they do not represent actors and dependencies among them. Moreover, we do not use tasks or resources that are important concepts in *i**. Our focus is on modeling and refinement of system goals and so we adopt the actor-less goal models consisting of one or more root goals that represent the objectives of the composite system – the system-to-be together with its environment. Similar goal model style is used in the formal approaches of [GMNS02] and [SGM04], as well as in KAOS approach.

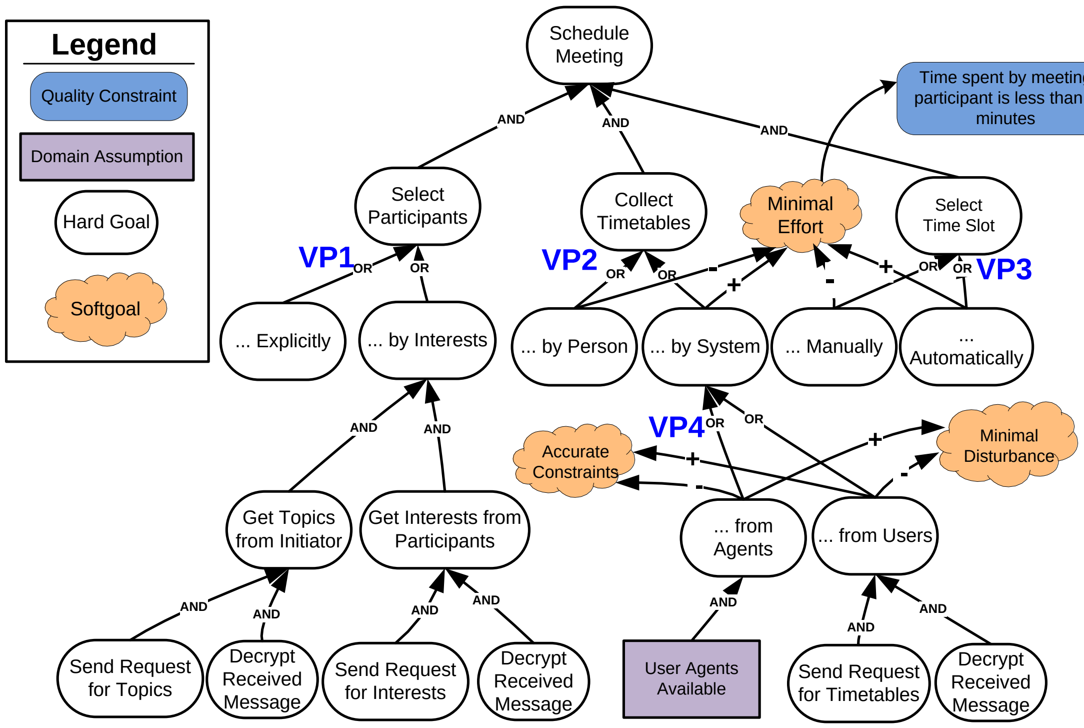


Figure 3-1. The Schedule Meeting goal model.

Let us suppose that the task is to design a system that supports scheduling meetings within an organization (Figure 3-1). *Goals* describe the states of the world that the stakeholders want to reach. Schedule Meeting in Figure 3-1 is an example of a *functional* or *hard* goal with a clear-cut satisfaction criterion. These goals are either fulfilled (satisfied) or not fulfilled (denied). It is easy to determine whether this particular goal is satisfied or not: if the meeting has been scheduled, then this goal has been satisfied. These goals are used to express functional requirements. They are represented by large ovals in the goal model. On the other hand, a goal like Minimal Effort is considered a *softgoal*, a goal with no clear-cut satisfaction criterion, which is hard to define formally. The reason this goal is a softgoal is because the evaluation of the amount of effort involved in scheduling meetings is open to individual interpretation, i.e., it is subjective. Similarly, it is not clear what constitutes the space of alternatives over which a "minimal effort" solution is to be chosen. We say that this type of goal can be *satisficed* – met to an acceptable degree. Softgoals are used to represent non-functional/quality requirements and are represented as the cloudy shapes in the models.

Goals and/or softgoals may be related through AND/OR relationships that have the obvious semantics (adopted from the planning domain in artificial intelligence) that AND-decomposed subgoals must all be attained for their parent goal to be achieved and at least one OR-decomposed subgoal needs to be achieved for achieving its parent goal. For example, to collect timetables from users, we need to send a request for these timetables AND decrypt the response. On the other hand, meeting participants can be selected either manually (one by one) or by their interests. The decomposition stops when we reach goals that can be delegated to the components of the system-to-be, to legacy systems, or to humans.

OR decompositions play a crucial goal in our approach. They allow capturing of the different ways that high-level goals can be gradually refined into solutions. In fact, the quantity of possible alternatives for attaining a top-level goal G grows exponentially with the introduction of a number of OR decomposition in the subtree representing the refinement of G as all combination of choices in all OR decompositions need to be considered to enumerate these alternatives. Thus, the OR decompositions in goal models allow us to represent *variability* in ways that stakeholder goals can be achieved, i.e. the variability in the problem domain. We call these *intentional* OR decompositions or *variation points* (VPs). They represent *intentional variability* in goal models. *Variability* is defined as all possible combinations of the choices in the variation points. For example, the four variation points of the goal model in Figure 3-1 are marked as "VP1" through "VP4". VP1 contributes two alternatives; VP2 and VP4 combined contribute 3, while VP3 contributes 2. Then, the total space of alternatives for this goal model includes 2*3*2 = 12 solutions.

In addition, goals/softgoals can be related to softgoals through help (+), hurt (–), make (++), or break (--) relationships. These *contribution links* allow alternatives represented in goal models to be evaluated with respect to important quality requirements captured by softgoals. For example, collecting timetables from users generally leads to more accurate constraints for scheduling meetings (see the positive contribution from Collect Timetables from Users to Accurate Constraints), but leads to the users being disturbed by the scheduling process. On the other hand, collecting timetables from user software agents may not be as accurate (e.g., if users' schedules are not up-to-date), but is better from the point of view of minimizing disturbance. Thus, alternative ways of attaining stakeholder goals can be ranked with respect to softgoals and different alternatives may be considered the "best" ones depending on which qualities are more important. In our example, if the softgoal Accurate Constraints has the priority over Minimal Disturbance, then the alternative with the manual timetable collection is preferred. The alternative that includes the collection of timetables from user agents is going to be selected in case Minimal Disturbance is preferred. This analysis can be done automatically with the help of the goal reasoning algorithm of [SGM04].

Softgoals can be refined into quality constraints, which are measurable non-functional requirements that describe qualities and constrain qualities [JMF08]. For instance, the softgoal Minimal Effort in the context of scheduling meetings can be refined into the quality constraint (QC) "the time spend by a meeting participant to schedule a meeting should be less than 5 minutes". Quality constraints may also be represented in the models (as dark ovals). The use of QCs in this thesis is confined to Chapter 8. Domain assumptions (DAs) capture the states of the world that we assume to be true. For instance, in order to be able to achieve the goal Collect Timetables from Agents (see Figure 3-1) there needs to be an assumption stating that user agents are available.

The relatively simple language presented above is usually sufficient for modeling and analyzing goals during requirements analysis, covering both functional and quality requirements. However, this framework abstracts over many properties of the problem domain that are of interest for the development of software systems. For example, there exist temporal or other dependencies among several goals that may require that one is achieved before the other. For instance, in Figure 3-1, the goal Send Request for Timetables must be achieved before the received message containing user timetables can be decrypted. On the other hand, the subgoals of Select Participants by Interest, namely Get Topics from the Initiator (i.e., get the topics of the upcoming meeting) and Get Interests from Participants, do not have any dependencies and thus can be achieved concurrently. The default goal model notation is unable to represent the above facts. In order to be able to represent this and other useful information we use goal model *annotations* (which were heavily used, for example, in [LL06]). Figure 5-1 presents a version of the Schedule Meeting goal model with a number of such annotations. In addition to the parallel (||) and concurrent (;) annotations, it captures whether OR decompositions are inclusive or exclusive. By default, OR decompositions in goal models are inclusive – any number of subgoals can be achieved for the parent goal to be achieved. However, in many cases it makes sense to make the decomposition exclusive using the “X” annotation. For instance, the selection of the time slot for a meeting should be done either manually (by the meeting initiator) or automatically (by the meeting scheduler system) since doing both will surely lead to confusion. The annotations that are used in goal models are selected based on the kind of information that one needs to capture. This is usually dictated by the type of analysis or transformations that the goal model is going to be subjected to. Thus, throughout this thesis we use varying types of goal model annotations, from control flow, OR/XOR, etc. in Chapter 5 to contextual annotations in Chapter 4. The key idea here is that these annotations are used to capture properties of the *problem* domain, not the solution domain. In addition to annotations, goal parameters representing, for example, the inputs required for the attainment of the goal and the outputs produced by the goal may be used. Thus, the goal Collect Timetables from Users may have the inputs “Users” and “Interval” and the output “Constraints”.

## 3.2 Reasoning with Goal Models

Two approaches are presented in [GMNS02] and [SGM04] for reasoning with goal models. Both approaches have the same formal setting. The first presents qualitative and numerical axiomatizations for goal models and introduces sound and complete label propagation algorithms for these axiomatizations. These algorithms work from the bottom to the top of goal models, in a fashion similar to the labelling algorithm in the NFR framework. In particular, given a goal model and labels for some of the goals, the algorithm propagates these labels towards the root goals. There is no distinction between functional goal and softgoals, so contribution links may be used with any pair of goals. Also, a new set of contribution links is introduced to capture the relationships where the contribution only takes place if a goal is satisfied. Goals in this framework have 4 possible values – satisfied, denied, partially satisfied, and partially denied. For more fine-grained analysis a quantitative variant of the approach is introduced. Here, the evidence for satisfiability/deniability of a goal is represented as a numerical value. The approach adopts a probabilistic model where the evidence of satisfiability/deniability of a goal is represented as the probability that the goal is satisfied/denied. On the other hand, in [SGM04] the authors attempt to solve a different problem: given a goal model, determine if there is a label assignment for leaf goals that satisfies or denies all root goals. A variation of the approach would find a minimum-cost label assignment that satisfies/denies root goals of the graph provided satisfaction/denial of every goal in the model has a certain unit cost. The approach works by reducing the problems to SAT and minimum-cost SAT for Boolean formulas. The above approaches provide a less-powerful, but much simpler alternative analysis method for reasoning about satisfaction/partial satisfaction of goals than [LL04]. The approach of [SGM04] provides the foundation of the analysis of goal models and the selection of suitable alternatives among the set of available ones.

Chapter 4
# Modeling Domain Variability in Requirements Engineering with Contexts

**Acknowledgement**: This chapter is based on [LM09].

## 4.1 Introduction

Domain models constitute an important aspect of requirements engineering (RE) for they constrain the space of possible solutions to a given set of requirements and even impact on the very definition of these requirements. In spite of that, domain models and requirements models have generally been treated in isolation by requirements engineering approaches (e.g., [DLF93]). As software systems are being used in ever more diverse and dynamic environments where they have to routinely and efficiently adapt to changing environmental conditions, their designs must support *high variability* in the behaviours they prescribe.

Not surprisingly, high variability in requirements and design has been recognized as the cornerstone in meeting the demands for software systems of the future (see Chapter 7 of this thesis as well as [LLYY06] and [DP90]). However, variability in domain models, which captures the changing, dynamic nature of operational environments for software systems, and its impact on software requirements, has not received equal attention in the literature. The problem is that traditional goal models assume that the environment of the system-to-be is mostly uniform and attempt to elicit and refine system goals in a way that would make the goal model adequate for the *most common* instances of a problem (e.g., selling goods, scheduling meetings, etc.) in a particular domain. In other words, traditional techniques ignore the impact of domain variability on the requirements to be fulfilled for a system-to-be. Thus, *these approaches are missing an important source of requirements variability*.

A recent proposal [SYN07] did identify the importance of domain variability on requirements. However, it assumes that requirements are given, and concentrates on making sure that they are met in every context. Thus, the approach does not explore the effects of domain variability on intentional variability – the variability in stakeholder goals and their refinements. Also, in pervasive and mobile computing, where contexts have long been an important research topic, a lot of effort has been directed at modeling various contexts (e.g., [HI04]), but little research is available on linking those models with software requirements [HCS05].

In a recent paper [LLYY06], Liaskos et al. concentrate on capturing *intentional* variability in early requirements using goal models. There, the main focus was on identifying all the ways stakeholder goals can be attained. We pointed out that non-intentional variability (that is, time, location, characteristics of stakeholders, entities in the environment, etc.) is an important factor in goal modeling as it constrains intentional variability in a significant way. However, we stopped short of systematically characterizing such domain variability and its effects on requirements. To that end, in this chapter, we propose a coherent process for exploring domain/contextual variability and for modeling and analyzing its effects on requirements models. We propose a fine-grained model of context that represents the domain state and where each context contains a partial requirements model representing the effects of that context on the model. Unlike, e.g., the method of Salifu et al. [SYN07], our approach results in *high-variability context-enriched goal models* that capture and refine stakeholder goals in *all* relevant contexts. These goal models can then be formally analyzed. Moreover, context refinement hierarchies and context inheritance allow incremental definition of the effects of contexts on goal models specified relative to higher-level contexts.

In addition, we can use this context framework not only to capture the *external context* of the system – i.e., things that are outside of the system, but also the *internal context*. Internal context encompasses things *within* the system that have an important effect on its behaviour. Internal faults are one example of what could be considered internal context. In Chapter 8, we use internal context hierarchies to capture and refine the goals of feedback controllers that arise in the presence of various errors and faults in the system.

As a motivation for this research, let us look at system for supplying customers with goods. This example is further discussed in Chapter 6. At a first glance, it seems that gathering requirements for such a system is rather straightforward: we have the domain consisting of the distributor, the customers, the goods, the orders, the shipping companies, etc. Following a goal-oriented RE approach, we can identify the functional goals that the system needs to achieve (e.g., Supply

Customer, see Figure 4-1) and the relevant softgoals like Minimize Risk and then refine them into subgoals until they are simple enough to be achieved by software components and/or humans. The produced requirements specification assumes that the domain is uniform – i.e., the specification and thus the system will work for *all* customers, *all* orders, etc. However, it is easy to see that this view is overly simplistic as it ignores the variations in the domain that have important effects on system requirements. E.g., international orders need to have customs paperwork filled out, while domestic orders do not. Large orders are good for business, so they may be encouraged by discounts or free shipping. And the list goes on. So, our aim in this thesis chapter is to introduce an approach that allows us to model these and other effects of domain non-uniformity and variability on software requirements.

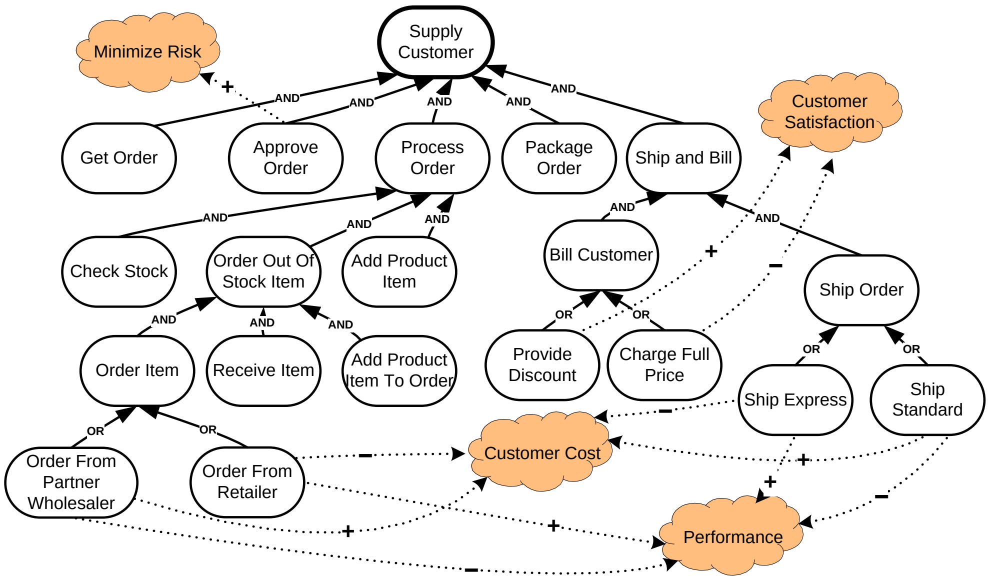


Figure 4-1. Supply Customer goal refinement.

## 4.2 Background: Contexts

There exist a lot of definitions of context in Computer Science. For example, Cappiello et al. [CCMP06] define context as "any information that can be used to characterize persons, places or objects that are considered relevant to the interactions between a user and an application, including users and applications themselves". Brezillon [Bre99] says that "context is what constrains problem solving without intervening in it explicitly". McCarthy states that "context is a generalization of a collection of assumptions" [MB97]. Both Brezillon's and McCarthy's definitions fit well with our treatment of context as properties of entities in the environment and of the environment itself that influence stakeholder goals and means of achieving them. Therefore, we define a context as an abstraction over a set of environment assumptions.

In various areas of computing, the notion of context has long been recognized as important. For example, in the case of context-aware computing the problem is to adapt to changing computational capabilities as well as to user behaviour and preferences [CCMP06]. In pervasive computing, context is used to model environment and user characteristics as well as to proactively infer new knowledge about users.

There have been quite a few recent efforts directed at context modeling. While some approaches adapt existing modeling techniques, others propose new or significantly modified notations. For instance, Henricksen and Indulska proposed the Context Modeling Language (CML) notation [HI04]. Their graphical modeling notation allows for capturing of fact types (e.g., *Located At*, *Engaged In*) that relate object types (e.g., location, person, and device). The model can distinguish between static and dynamic facts. Moreover, it classifies dynamic facts into profiled facts (supplied by users), sensed facts (provided by sensors), and derived facts (derived from other facts through logical formulas). Dependencies among facts can also be specified. A special temporal fact type can be used to capture time-varying facts. Additional features of the approach include, for example, support for ambiguous context as well as for context quality (e.g., certainty).

Standard modeling approaches like UML and ER have also been used for context modeling. For example, in [CDFM07], UML class diagrams are used to model user, personalization, and context metadata subschemas together in one model. However, these models are not well suited for capturing certain special characteristics of contextual information [HI04]. In addition, ontologies are also used for context modeling. They provide extensibility, flexibility and composability for contexts. In [CCMP06], a generic top ontology, which can be augmented with domain-dependent ones, is proposed. However, these approaches do not focus on the use of context in applications.

## 4.3 Related Work

Much research has been dedicated to the formal handling of contexts in the area of Artificial Intelligence and Knowledge Representation and Reasoning. These approaches can be split into two types [BGGB03]. First is *divide-and-conquer*, which assumes that there is a global theory of the world. The structure of that theory is articulated into a collection of contexts. There is a single representation language and different subsets of the global model are “localized” into contexts. An alternative view is called *compose-and-conquer* and states that there is no global theory of the world and that each context is a viewpoint on the world, possibly expressed in its own language. There exist methods combining the two views, such as [MM95], where contexts are used to partition knowledge bases while changes can be propagated across contexts. Our proposal and the formal framework presented in Section 4.4 fits within the divide-and-conquer paradigm since it is designed to support the capturing of context-dependent model variations.

Our view of contexts is somewhat similar to that of the CYC common sense knowledge base [Len98]. CYC has 12 context dimensions along which contexts vary. Each region in this 12-dimensional space implicitly defines an overall context for CYC assertions. We, however, propose more fine-grained handling of context with possibly many more domain-specific dimensions.

Brezillon et al. [BPP02] propose an approach for modeling the effects of context on decision making using contextual graphs. Contextual graphs are based on decision trees with event nodes becoming context ones. They are used to capture various context-dependent ways of executing procedures. Whenever some context plays a role in the procedure, it splits it up into branches according to the value of the context. Branches may recombine later. This work is close to our approach in the sense that it attempts to capture all the effects of contextual variability in one model. However, we are doing this at the intentional level, while contextual graphs is a process-level notation. Moreover, it does not consider quality attributes. We have looked at generating process-level specifications from goal models [LYM07] and we believe that contextual graphs can be generated from the context-enriched goal models as well.

Salifu et al. [SYN07] suggest a Problem Frames-based approach for the modeling of domain variability and for the specification of monitoring and switching requirements. They identify domain variability (modeled by a set of domain variables) using variant problem frames and try to assess its impact on requirements. For each context that causes the requirements to fail, a variant frame is created and analyzed in order to ensure the satisfaction of requirements. This approach differs from ours in that it assumes that the requirements specification is given, while we are concentrating on activities that precede its formulation. Another substantial difference is that we propose the use of a single high-variability goal model for capturing all of the domain's variability.

## 4.4 The Formal Framework

In this section, we present a formal framework for managing models through the use of contexts. While we are mainly interested in the graphical models such as goal models, our approach equally applies to any type of model, e.g., formal theories. We view instances of models (e.g., the Supply Customer goal model) as collections of model element instances (e.g., Ship Order). There may be other important structural properties of models that need capturing, but we are chiefly concerned with the *ability to model under which circumstances certain model elements are present (i.e., visible) in the model* and with the ability to display a version of the model for the particular set of circumstances. Thus, we are concerned with capturing *model variability* due to a wide variety of *external factors*. These factors can include viewpoints, model versions, domain assumptions, etc. This formal framework can be instantiated for any model to help with managing this kind of variability. In Section 4.5.3, we present an algorithm that generates this formal framework given an instance of a goal model.

We assume that there are different types of elements in a modeling notation. For example, in graphical models, we have various types of nodes and links among them. Let **M** be the set of model element instances in a model. Let **T** be the set of various model element types available in a modeling notation (e.g., goals, softgoals, etc.). The function $L$ maps each element of **M** into an element of **T**, thus associating a type with every model element instance. Only certain types of elements in a modeling notation may be affected by contexts and thus belong to a variable part of a model. We define $\mathbf{T^C}$ as the subset of **T** containing such *context-dependent* model element types. If a model element type is not in $\mathbf{T^C}$, it is excluded from our formalization. The contents of the $\mathbf{T^C}$ set are notation and model-dependent. Let $\boldsymbol{M^C} \overset{\text{def}}{=} \{n |\ n \in \boldsymbol{M}\ \wedge L(n) \in \boldsymbol{T^C}\}$ be the set of modeling elements of the types that can be affected by contexts.

We next define the set **C** of *contextual tags*. These are labels that are assigned to model elements to capture the conditions that those elements require to be visible in the model. To properly define what contextual tags model, we assign each tag a Boolean expression that specifies when the tag is *active*. Since the tags represent domain properties, assumptions, etc., the associated expressions precisely define when the contextual tags affect the model and when they are not (we define $\mathbf{P}$ to be the set of Boolean expressions): $active\colon \boldsymbol{C} \rightarrow \boldsymbol{P}$**.** For example, the tag *largeOrder* describes a real world entity and may be defined as an order with the overall price of all goods being over \$10K. So, when some order *is* over \$10K, the tag becomes active and thus can affect the model. The approach can also be used to capture viewpoints, model versions, etc. In those cases, the definition of tags can be simple: they can be turned on and off depending on what the modeler is interested in (e.g., *versionOne* = *true*). We also allow negated tags to be used in the approach: $(\forall t \in \boldsymbol{C})\ active(\neg \mathrm{t}) \stackrel{\text{def}}{=} \neg active(t)$.

We define a special *default* tag that is always active and if assigned to an element of a model signifies that the element does not require any assumptions to hold to be present in the model. To associate tags with model elements we create a special relation called *taggedElement* ($\wp$ is a powerset): $taggedElement\ \subseteq\ \boldsymbol{M^c} \times\ \wp(\wp(\boldsymbol{C}))$.

To each element of $\mathbf{M^C}$ we assign possibly many tag combinations (sets of tags). E.g., the set *{{a,b},{c,d}}* assigned to an element *n* specifies that *n* appears in the model in two cases: when both *a* and *b* are active or when both *c* and *d* are active. The outer set is the set of alternative tag assignments, either of which is enough for the element to be visible. In fact, the above set can be interpreted as $(a \wedge b) \vee (c \wedge d)$, so our set of sets of tags can be viewed as a propositional DNF formula.

The function *newTaggedElement* creates a new tagged element entity given a model element and a set of tags. It can be called from within algorithms that process input models for which we want to use the formal framework. Given a model element, the function *tags* returns the set of contextual tags of a *taggedElement* (it is defined as $tags\colon taggedElement\ \rightarrow\ \wp(\wp(\boldsymbol{C}))$. In order to eliminate possible inconsistent sets of tags (i.e., having both a tag and its negation) from the set returned by *tags(n)*, we define the following set for each model element:

$$context(n) = \{K | K \in tags(n) \wedge \neg inconsistent(K)\}.$$

## 4.4.1 Inheritance of Contextual Tags

Contextual tags can *inherit* from other tags (no circular inheritance is allowed). This is to make specifying the effects of external factors on models easier. E.g., we have a tag *substantialOrder* applied to certain elements of a business process (BP) model. Now, we define a tag *largeOrder* inheriting from *substantialOrder*. Then, since *largeOrder is-a substantialOrder*, the derived tag can be substituted everywhere for the parent tag. Thus, the elements that are tagged with *substantialOrder* are effectively tagged with *largeOrder* as well. Of course, the converse is not true. Apart from being automatically applied to all the elements already tagged by *substantialOrder*, we can explicitly apply *largeOrder* to new nodes to specify, for example, that the goal Apply Discount requires large orders. The benefits of contextual tag inheritance include the ability to reuse already defined and applied tags and thus to develop context-dependent models incrementally. We state that one tag inherits from another by using the predicate *parent(parentTag,childTag)*. *Multiple inheritance* is allowed, so a tag can inherit from more than one parent tags. In this case, the derived tag can be used in place of all of its parent tags, thus inheriting the elements tagged by them. *parent* is extensionally defined based on the contextual tag inheritance hierarchy associated with the source model. *ancestor(anc,dec)* is defined trough *parent* to indicate that the tag *anc* is an ancestor of *dec*.

We also support a simple version of *non-monotonic inheritance* where certain elements tagged by an ancestor tag may not be inherited by the derived tag. Suppose the goal Apply Shipping Discount is tagged with *substantialOrder*, i.e., it applies to substantial (large and medium) orders only. However, we might not want this goal to apply to large orders (as it would with regular inheritance) since we want them to ship for free. So, we declare this model element *abnormal* with respect to the inheritance of *largeOrder* from *substantialOrder* and that particular activity, which means that the *largeOrder* tag will not apply to it. We can do this by using the following: *ab(dec,anc,n)*, where *dec*, *anc* $\in \mathbf{C}$ and $n \in \mathbf{M}^{\mathbf{C}}$. This states that for the element *n* the descendent contextual tag (*dec*) cannot be substituted for the ancestor tag (*anc*). In fact, given tag combinations applied to *n*, we can determine if it is abnormal with respect to some inheritance hierarchy if there is a tag combination with an ancestor tag and a negation of a descendent tag:

$$ab\_base(dec, anc, n) \overset{\text{def}}{=} \big(\exists K \in context(n)\big)\, ancestor(anc, dec) \wedge anc \in K \wedge \neg dec \in K$$

If a context *dec* is found to be abnormal with respect to one of its ancestors *anc* and a node *n*, then all of *dec*'s descendents are also considered abnormal with respect to that node. So, in general, a context *dec* is viewed as abnormal with respect to its ancestor context *anc* and a model node *n* if it either appears negated in the same tag combination with (non-negated) *anc* (i.e., this is the base case, as above) or *dec* is a descendent of another contextual tag *int* (intermediate context) that is a descendent of *anc* and appears negated in the same tag combination as *anc* (i.e., the base case applies to *int*):

$$ab(dec, anc, n) \stackrel{\text{def}}{=}$$
$$ab\_base(dec, anc, n) \vee (\exists int \in \boldsymbol{C}) ancestor(int, dec) \wedge ab\_base(inc, anc, n)$$

### 4.4.2 Visibility of Modeling Elements

Given the sets of contextual tags applied to context-dependent model elements and the formulas defining when those tags are active, we can determine for each such element whether it is visible in the model. We define the following function:

$$visible\colon \boldsymbol{M^C} \rightarrow \{true, false\}$$

$$visible(n) \stackrel{\text{def}}{=} \big(\exists K \in context(n)\big)(\forall e_i \in K)$$

$$\bigwedge\nolimits_{1 \le i \le |K|} active(e_i) \vee \exists d\ (ancestor(\,e_i, d) \wedge \neg ab(d, e_i, n) \wedge active(d))$$

Thus, we define a context-dependent model element to be visible in a model if there exists a contextual tag assignment $K$ for that element where each tag is either active itself or there exists its active non-abnormal descendent tag. Now we can produce the definition of the subset of visible context-dependent elements of a model: $\boldsymbol{V} \stackrel{\text{def}}{=} \{n | n \in \boldsymbol{M^C} \wedge visible(n)\}$. Note that for most modeling notations we also need other (e.g., structural) information in addition to the set **V** to produce a valid submodel corresponding to the current context. Since that information is notation-dependent, it is not part of our generic framework. Also note that since the definitions of contextual tags likely refer to the real-world phenomena, if the approach is used at runtime, the visibility of model elements can dynamically change from situation to situation.

## 4.5 Contextual Variability in Goal Modeling

In this section, we introduce our approach for modeling and analysing the effects of context on requirements models. We use the Supply Customer case study (see Figure 4-1), which is a variation of the one discussed in Chapter 6.

Our method involves a number of activities. Some of these activities are discussed in the subsequent sections, while here we outline the approach:

1. Identify the main purpose of the system (its high-level goals) and the domain where the system is to operate.
2. *Iterative step*. Refine the goals into lower-level subgoals.
3. *Iterative step*. Identify the entities in the domain and their characteristics that can affect the newly identified goals. Capture those effects using *contextual tags*. Update the context model.
4. Generate the formal model for managing context-dependent variability.
5. Analyze context-enriched goal models
   a. Given currently active context(s), produce the corresponding goal model.
   b. Analyze whether top-level system goals can be attained given currently active context(s). The standard goal reasoning techniques can be applied since *the contextual variability has been removed*.

## 4.5.1 Context Identification and Modeling

Our goal in this approach is to systematically identify domain variability and its effect on stakeholder goals and goal refinements. Unlike intentional variability discussed in [LLYY06], domain variability is external to the requirements model, but influences intentional variability and thus requirements. We represent domain models in terms of *contexts* – properties or characteristics of the domain that have effect on requirements – and thus variability in the domain is reflected in the contextual variability.

Note that there may be certain aspects of the domain that do not affect requirements and these are not important to us. *Context entities*, such as actors, devices, resources, data items, etc., are things in the domain that influence the requirements (e.g., an *Order* is a context entity). They are the *sources of domain variability*. We define a context entity called *env* for specifying ambient properties of the environment. A *context variability dimension* is an aspect of a domain along which that domain changes. It may be related to one or more context entities (e.g., *size(Order)* and *relativeLocation(Warehouse,Customer)*). A dimension can be thought of as defining a range or a set of values. A *context* is a particular value for a dimension (e.g., *size(Order,<u>$5000</u>)*, *relativeLocation(Warehouse,Customer,<u>local</u>)*).

Figure 4-2B shows the metamodel that we use for capturing the basic properties of domain variability (such as context entities and variability dimensions) in our approach. Additional models can also be useful. As mentioned in Section 4.2, there are a number of notations that can be employed for context modeling. Figure 4-2A presents a UML class diagram variation showing the context entities in our case study (their corresponding context dimensions are modeled as attributes). In addition to UML or ER diagrams for context modeling, specialized notations like the CML are able to specify advanced properties of contexts (e.g., derived contexts).

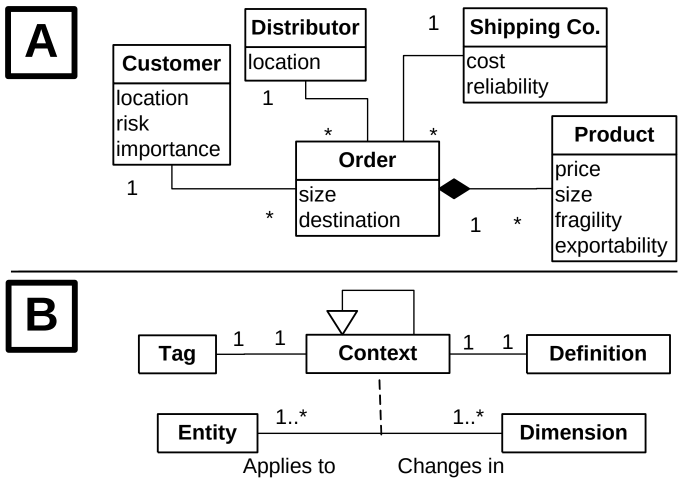


Figure 4-2. UML context model for the case study (A) and our context metamodel (B).

Unlike the simpler notion of context in CML and in some other approaches, we are proposing the use of *context refinement hierarchies* for the appropriate context dimensions. Their purpose is twofold: first, they can be used to map the too-low-level contexts into higher-level ones that are more appropriate for some particular application (e.g., GPS coordinates can be mapped into cities and towns). This is commonly done in existing context-aware applications in the fields such as mobile and pervasive computing. Second, abstract context hierarchies may be useful in terms of splitting contexts into meaningful, appropriately named high-level ranges. For example, an order size (in terms of the dollar amount) is a number. So, one can specify the effects of orders of over $5,000 on the achievement of the subgoal Approve Order, then orders over $10,000, etc. However, very frequently, and *especially during requirements elicitation and analysis*, it is more convenient to specify what effect certain *ranges* of context have on goal models. For example, instead of thinking in terms of the dollar amounts as in the example above, it might be more convenient to reason in qualitative terms like *Large Order* or *Medium Order* (see Figure 4-3A, where *Size* is the context dimension of the *Order* context entity, while the arrows represent IS-A relationships among contexts and the boxes capture the possible contexts in the hierarchy). The high-level contexts will need to be refined into lower-level ones and eventually defined using the actual order amounts. We call such defined contexts *base contexts* (note the "B" label on the leaf-level contexts in Figure 4-3).

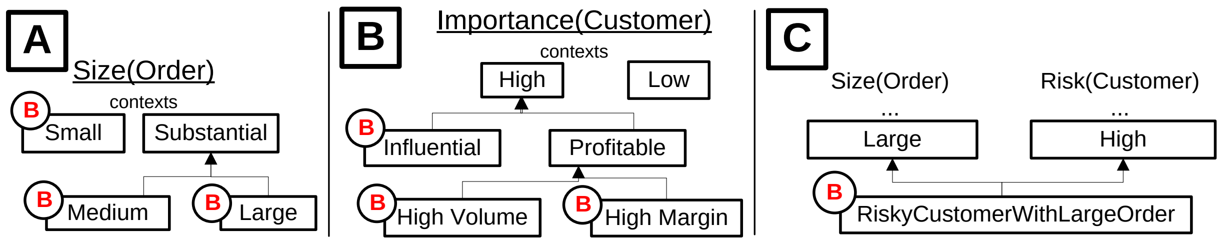


Figure 4-3. Order size (A) and Customer importance (B) context hierarchies and multiple inheritance (C).

A context must be defined through a *definition*, a Boolean formula, which is specified using the expression of the type *Dimension(Entity(-ies),Context)* $\stackrel{\text{def}}{=}$ *definition*. If it holds (i.e., the domain is currently in the state defined by the context), we call that context *active*. For example, large orders may be defined as the ones over \$1000. Thus, formally: *size(Order, large)* $\stackrel{\text{def}}{=} \exists n$ *size(Order, n)* $\wedge\ n \geq \$1000$. As mentioned before, contexts may have concrete definitions or may be defined through their descendant contexts: *size(Order, substantial)* $\stackrel{\text{def}}{=}$ *size(Order, large)* $\vee$ *size(Order, medium)*. There should be no cycles in context dependencies.

Contexts may be derived from several direct ancestors, thus inheriting their effects on the goal model. In Figure 4-3C, we create a new context by deriving it from the contexts *size(Order,large)* and *risk(Customer,high)*. This produces a new context dimension with both context entities becoming its parameters. We also need to provide the definition for the new context, i.e., to specify when it is active: *sizeRisk(Customer,Order, riskyCustomerWithLargeOrder)* $\stackrel{\text{def}}{=}$ *size(Order,large)* $\wedge$ *risk( Customer, high)*. Thus, it is active precisely when the customer is risky and the order is large.

While context refinement hierarchies provide more flexibility for handling contexts, their design should not be arbitrary. When developing context hierarchies in our approach, care must be taken to ensure that they are not unnecessarily complicated, i.e., that the contexts are actually used in goal models.

### 4.5.2 Modeling the Effects of Contextual Variability on Goal Models

In Section 4.5.1, we discussed the modeling of domain characteristics using contexts. Here, we show how the effects of domain variability on requirements models can be captured. The idea is to be able to model the effects of all relevant contexts (i.e., the domain variability) conveniently in a single model instance and to selectively display the model corresponding to particular contexts. We use *contextual tags* (as in Section 4.4) attached to model elements to visually specify the effects of domain variability on goal models. While context definitions and inheritance hierarchies make up the domain model, we need to specify how contexts affect the graphical models, i.e., which elements of the models are *visible* in which contexts.

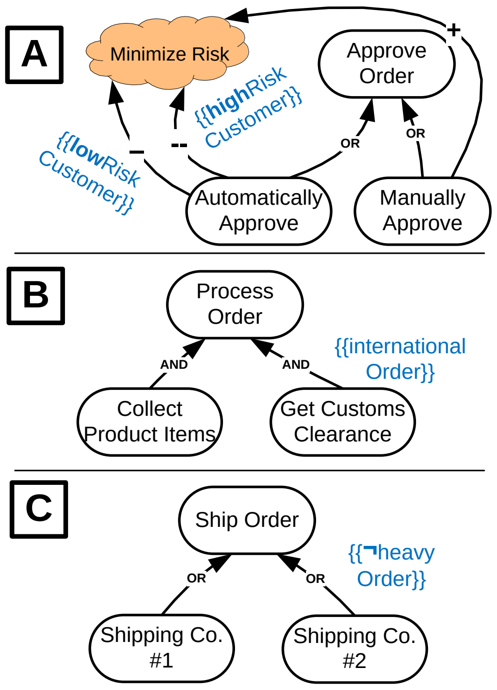


Figure 4-4. Specifying effects of domain variability using contextual tags.

#### 4.5.2.1 Effects of Contexts on Goal Models

Domain variability can influence a goal model in a number of ways. Note from the following that it can only affect (soft)goal nodes and contribution links. Domain variability affects:

- *The requirements themselves*. (Soft)goals may appear/disappear in the model depending on the context. For instance, if a customer is willing to share personal details/preferences with the seller, the vendor might acquire the goal Up-sell Customer to try and sell more relevant products to that customer.

- *The OR decomposition of goals*. New alternative(s) may be added and previously identified alternative(s) or may be removed in certain contexts. For example, there may be fewer options to ship heavy orders to customers (Figure 4-4C).

- *Goal refinement*. For example, the goal of processing an international order is not attained unless the customs paperwork is completed (Figure 4-4B). This, of course, does not apply to domestic orders.

- *The assessment of various choices in the goal model*. E.g., automatic approval of *orders* from low-risk customers may hurt ("–") the Minimize Risk softgoal, while doing the same for very risky ones will have a significantly worse ("--") effect on it (Figure 4-4A).

#### 4.5.2.2 Effects Identification

When applying the approach, the activities of developing contextual models and the identification of the effects of contexts on goal models need to proceed iteratively. While it is possible to attempt to identify all the relevant context entities and their dimensions upfront, it is very likely that certain important dimensions will be overlooked. For example, after the modeler refines the goal Package Order enough (see Figure 4-1), he/she will elicit the goal Package Product. Only after analyzing which properties of a product can affect its packaging, will the modeler be able to identify the dimension *Fragility* as relevant for the context entity *Product*. Therefore, to gain the maximum benefit from the approach, the activities of context modeling need to be interleaved with the development of context-enriched goal models. Thus, the context model will be gradually expanded as the goal model is being created.

In our approach, when refining a goal, we need to identify the relevant context entities and their context dimensions that may influence the ways the goal is refined. There are a number of ways such relevant context entities can be identified. For example, in some versions of the goal modeling notation, goals have parameters (e.g., Process Order(Customer,Order), as discussed in Chapter 6 of this thesis), which are clearly context entities since their properties influence the way goals can be attained. Alternatively, a variability frame of a goal [LLYY06] can be a powerful tool for identifying relevant context entities and dimensions for a goal. We can use a table to document *potentially* relevant context entities (columns) and their dimensions (rows) for goals. While certain entities and/or dimensions currently may have no effect on the refinement of the goal, it is still prudent to capture them for traceability and future maintenance. For instance, below is the table where we identified order size and destination as well as customer importance as dimensions affecting the goal Apply Discount.

Table 4-1. Context dimensions affecting the goal Apply Discount.

| *Apply Discount* | Entity: Order | Entity: Customer |
|---|---|---|
| Dimensions | Size, Destination | Importance |

### 4.5.2.3 Specifying the Effects of Contexts on Goal Models

Tags are mnemonic names corresponding to contexts. For example, *largeOrder* may be the tag for the context *size(Order,large)*. Contextual tags are applied to model elements to specify the effects of domain variability on goal models – i.e., to indicate that certain contexts are required to be active for those elements to be visible in the model. As in Section 4.4, we have sets of alternative tag assignments and all the tags within any such assignment must be active for the model element to be visible. E.g., the set of tags *{{largeOrder}, {importantCustomer, mediumOrder}}* attached to the goal Apply Discount indicates that ether the order has to be large or there must be an important customer with a medium-sized order to apply a discount. *Not* (¬) can be used with tags to indicate that the corresponding context cannot be active if the node is to be visible (see Figure 4-4C).

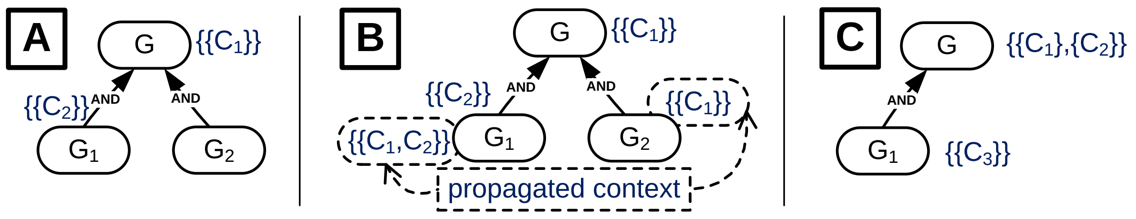


Figure 4-5. Contextual tag assignment examples.

By default, model elements are said to be contained in the *default context*, which is always active (*{{default}}*). To specify that the goal G must only be achieved when the context $C_1$ is active, we apply the tag *{{$C_1$}}* to G (Figure 4-5A). If we want a goal to be achieved when either of contexts is active, several sets of tag assignments must be used. E.g., the tag *{{$C_1$},{$C_2$}}* applied to G (Figure 4-5C) indicates that $C_1 \vee C_2$. When a set of tags is applied to the goal node G, it is also applied (implicitly propagated, see Figure 4-5B) to the whole subtree rooted at that goal. The hierarchical nature of goal models allows us to drastically reduce the number of contextual tags used in the model. Tag sets are combined when used in the same goal model subtree. E.g., if a tag set *{{$C_2$}}* is applied to the node $G_1$ in the subtree of G (Figure 4-5B), then $G_1$ (and thus the subtree rooted at it) is to be attained only when *both* contexts corresponding to $C_1$ and $C_2$ are active, which is indicated by the tag *{{$C_1$,$C_2$}}* (i.e., $C_1 \wedge C_2$). The tags applied to G and $G_1$ (Figure 4-5C) when combined produce *{{$C_1$,$C_3$},{$C_2$,$C_3$}}* since $(C_1 \vee C_2) \wedge C_3 = (C_1 \wedge C_3) \vee (C_2 \wedge C_3)$. The above also applies to softgoals.

## 4.5.3 Analyzing Context-Dependent Goal Models

In Section 4.4, we presented a generic formal framework for handling context-dependent models. It provides the basis for managing model variability due to external factors such as domain assumptions, etc. Here, we show how the formal framework can be used together with goal models to analyze domain variability in requirements engineering. In order to use the framework with goal models, we need a procedure that processes these models together with context inheritance hierarchies and generates the required sets and facts for the formal framework to operate on.

There are several steps in the process of generating the formal framework for goal models. First, we create the *parent* facts that model the tag inheritance hierarchy based on the context hierarchies described in Section 4.5.1. Similarly, definitions of the contexts will be assigned to the corresponding contextual tags and will be returned by the *active(context)* function for evaluation to determine if these tags are active.

We then state which elements of goal models we consider to be context-dependent. In general, the set $\mathbf{T}^{\mathbf{C}}$ = {G (goals), S (softgoals), R (contribution links)}. Below is the algorithm that completes the creation of the formal framework: it traverses the goal model and generates *taggedElement* instances corresponding to the context-dependent elements of the model along with the sets of tags assigned to these elements.

The procedure *generateContextModel* takes the set of root (soft)goals as the input and calls the procedure *processNode* on the (potentially) many (soft)goal trees that comprise the goal model. *processNode* has two parameters: the node *e* being processed and the set of tag assignments from the parent node, *pC* (parent context). Since we start from the root goals, initially *pC* has the value *{{default}}*. Within the *processNode* procedure we first check if the node *e* has a set of contextual tags *A* attached. If it does, it means that we must combine the parent context *pC* with *A* to produce the complete set of tags for *e*. If *pC* is the default context, it will simply be replaced by the tag set *A*. Otherwise, both *pC* and *A* are combined (as described in Section 4.5.2.3) to produce the new set of tags for *e* (see lines 7-11). We create the *taggedElement* unit for *e* with the newly produced context in line 16. The softgoal contribution links emanating from and terminating at *e* are processed by the *processLinks* function that computes the tag assignment for the links. Note that *newContext* is provided to *processLinks* as it becomes its parent context. Then we recursively process all the child nodes of *e* providing *newContext* as their parent context.

**Algorithm 1**: *Formal model generation*

Input: a set *O* of root (soft)goals of a goal model

Output: a formal model in the notation described in Section 4.4

```
1:  procedure generateFormalModel($O$)
2:    for each $e \in O$ do
3:      processNode($e$, $\{\{default\}\}$)
4:    endFor
5:  endProcedure
```

**Algorithm 2**: *Traverse goal model*

Input: element *e* and its parent context *pC*

Output: *taggedElement* entities in the formal model

```
01: procedure processNode($e$, $pC$ )
02:   $newContext \leftarrow \varnothing$
03:   if context annotation $A$ exists for $e$ then
04:     if $pC = \{\{default\}\}$ then
05:       $newContext \leftarrow A$
06:     elseIf //parent context is non-default
07:       for each $K_1 \in pC$ do
08:         for each $K_2 \in A$ do
09:           $newContext \leftarrow newContext \cup \{K_1 \cup K_2\}$
10:         endFor
11:       endFor
12:     endIf //default context
13:   elseIf  //annotation does not exist
14:     $newContext \leftarrow pC$
15:   endIf //annotation
16:   newTaggedElement($e$, $newContext$)
17:   $processLinks(e, newContext)$
18:   for each child (soft)goal node $c$ of $e$ do
19:     $processNode(c, newContext)$
```

```
20:   endFor
21: endProcedure
```

**Algorithm 3**: *Process contribution links*

Input: (soft)goal node *e* and its context *eContext*

Output: *taggedElement*s for *e*'s contribution links

```
01: procedure processLinks(e, eContext)
02:   for each outgoing contribution link l of e do
03:     linkContext ← ∅ //stores link context
04:     if context annotation $A_l$ exists for l then
05:       if eContext = {{default}} then
06:         linkContext ← $A_l$
07:       elseIf //node context is non-default
08:         for each $K_1$ ∈ eContext do
09:           for each $K_2$ ∈ $A_l$ do
10:             linkContext ← linkContext ∪ {$K_1$ ∪ $K_2$}
11:           endFor
12:         endFor
13:       endIf //default context
14:     elseIf //annotation does not exist
15:       linkContext ← eContext
16:     endIf //annotations
17:     if taggedElement(l) exists then
18:       for each $K_1$ ∈ taggedElement(l).tags do
19:         for each $K_2$ ∈ linkContext do
20:           taggedElement(l).tags ←
                  taggedElement(l).tags ∪ {$K_1$ ∪ $K_2$}
21:         endFor
22:       endFor
23:     elseIf //taggedElement does not exist
24:       newTaggedElement(l, linkContext)
25:     endIf
26:   endFor //processing of outgoing links
27:   for each incoming contribution link l of e do
28:     if taggedElement(l) exists then
29:       if eContext ≠ {{default}} then
30:         for each $K_1$ ∈ taggedElement(l).tags do
31:           for each $K_2$ ∈ eContext do
32:             taggedElement(l).tags ←
                    taggedElement(l).tags ∪ {$K_1$ ∪ $K_2$}
33:           endFor
34:         endFor
35:       endIf //non-default eContext
36:     elseIf //taggedElement(l) does not exist then
37:       newTaggedElement(l, eContext)
38:     endIf //taggedElement(l) does not exist
39:   endFor //processing of incoming links
40: endProcedure
```

The visibility of contribution links is determined by three contexts: the context attached to the link itself through an annotation (if present) and the contexts associated with both its source and destination nodes. All of these three contexts have to be active for the link to be visible. Associating such a combined context expression with contribution links is what the *processLinks* procedure does. The key here is that when processing a contribution link that is outgoing from the node *e*, we combine its annotation and the context associated with the source node *e*. On the other hand, when processing an incoming link, we only add the context from the target node to the link's context assignment. In particular, in lines 2-26, *processLinks* looks at contribution links outgoing from the node *e*. When processing outgoing links, we combine the context from the source node, *eContect*, and the context annotation that is possibly applied to the contribution link. We first construct the new context (*linkContext*) for the contribution link (lines 2-16). If there exists a context annotation applied to the link, then this combination is done in a way that is similar to how we combined contexts in *processNode* (lines 4-13). Otherwise, the context from the source node becomes the new link context. Once the context is computed, a new *taggedElement* is created for the link model element (line 24) if it does not exist already (there is a possibility that we have already processed the target node of the link *l* and thus have already created the appropriate *taggedElement* entity). When processing incoming contribution links, we combine the context applied to their target nodes (*eContext*) with the tag assignment that already exists for the link's *taggedElement*. If *taggedElement* does not yet exist, we create it and put *eContext* as the link's contextual tag assignment. When we process the source node of the contribution link, the context assignment will be completed.

After *generateFormalModel* and other mapping procedures have been executed, we have a formal context framework that can be used to produce the set of elements visible in the model in the current context. Using Figure 4-6 we illustrate how contexts can be used to generate context-specific models from high-variability context-enriched goal models. The main idea here is that a context-enriched goal model captures the effects of domain variability on requirements. Knowing which contexts are currently active and which are not (this, in effect, binds the contextual/domain variability), we can produce a goal model for that *particular domain variation*. This is done by removing the model elements, which are not visible in the currently active context. For example, in Figure 4-6A, we have a generic context-enriched goal model with three contexts (none of the contexts belong to the same inheritance hierarchy or defined through each other for simplicity) and a number of contextual annotations applied to the model. Figure 4-6B shows this goal model in the domain where all of the contexts are active. Only the subtree that is tagged with the ¬*2* context is removed (the removed portion of the model is greyed out).

In Figure 4-6C, only the context *1* is active, which results in two nodes being removed. Figure 4-6D shows the model in the domain where only context *3* is active.

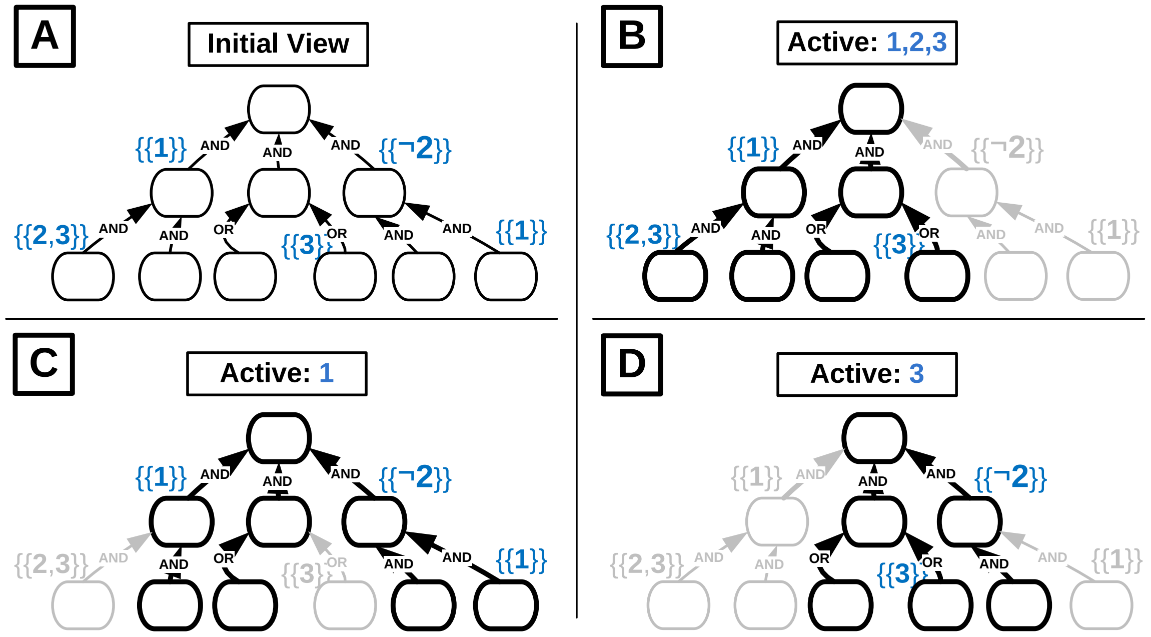
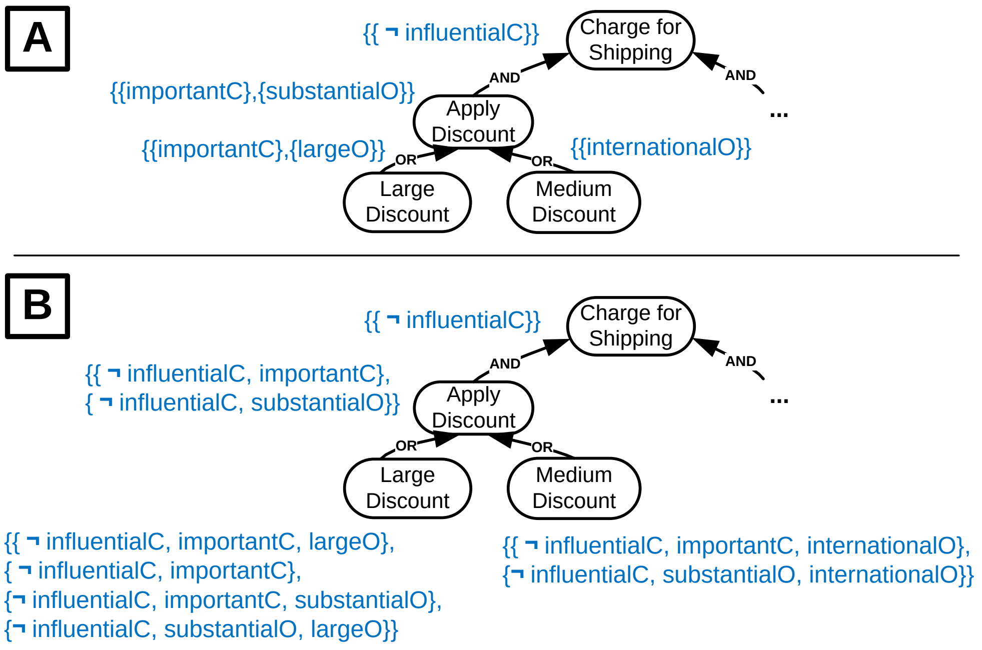


Figure 4-6. Using contexts as filters on goal models.

We also use a portion of our case study to further demonstrate the binding of contextual variability as well as the analysis that can be performed on context-enriched goal models with the aid of our approach. Figure 4-7A shows a fragment of the process Supply Customer for calculating shipping charges. Influential customers are not charged for shipping, so the context *{{¬influentialC}}* is applied to it. Note that in the contextual tags used in Figure 4-7 and the subsequent figures in this section, "C" stands for "Customer", while "O" stands for "Order". We apply discounts only in cases involving important customers or substantial orders, so Apply Discount is tagged with *{{importantC},{substantialO}}*. [Provide] Large Discount is tagged with *{{importantC},{largeO}}*: it applies to large orders or to important customers. Finally, Medium Discount applies to international orders only. The complete sets of contextual tags applied to each node are calculated (Figure 4-7B) by appropriately propagating contextual tags applied to the ancestor nodes.

Figure 4-7C shows a fragment of the formal model generated by the algorithm presented earlier. The portion of the formal model shown deals with the context inheritance (the *ancestor* relations) and with the abnormality of contexts with respect to certain model elements. We use the inheritance hierarchies from Figure 4-3. Note that the influential customer context tag (*influentialC*) is found to be abnormal with respect to important customers (*importantC*) in the subtree Apply Discount. The reason for this abnormality is the fact that the top goal in this model fragment, Charge for Shipping, is tagged with ¬*influentialC*, while its subgoal Apply Discount is tagged with *importantC*, which is a descendent context of *influentialC*. By definition of abnormality (see Section 4.4.1), these tags are found to be abnormal with respect to the subtree rooted at Apply Discount.


A
{{ ¬ influentialC}}
Charge for Shipping
AND
AND
{{importantC},{substantialO}}
Apply Discount
...
{{importantC},{largeO}}
OR
OR
{{internationalO}}
Large Discount
Medium Discount
B
{{ ¬ influentialC}}
Charge for Shipping
AND
AND
{{ ¬ influentialC, importantC},
{ ¬ influentialC, substantialO}}
Apply Discount
...
OR
OR
Large Discount
Medium Discount
{{ ¬ influentialC, importantC, largeO},
{ ¬ influentialC, importantC},
{¬ influentialC, importantC, substantialO},
{¬ influentialC, substantialO, largeO}}
{{ ¬ influentialC, importantC, internationalO},
{¬ influentialC, substantialO, internationalO}}
C

| **Ancestor:** | **Abnormality**: |
|---|---|
| ancestor(importantC,influentialC) | ab(influentialC,importantC,Apply Discount) |
| ancestor(importantC,highVolumeC) | ab(influentialC,importantC,Large Discount) |
| ancestor(substantialO,largeO) | ab(influentialC,importantC,Medium Discount) |
| ancestor(substantialO,mediumO) | |

Figure 4-7. The initial, context-annotated goal model fragment (A), the complete propagated tag assignment (B), and the inheritance and abnormality portion of the formal context model (C).

By using context definitions (not shown), we can determine which contextual tags are active and thus affect the model. Suppose that we are in the context of a large international order (Figure 4-8A). The ¬*influentialC* tag is active in this case, so Charge for Shipping is visible. Apply Discount is too since a large order IS-A substantial order and so both tags in *{¬influentialC, substantialO}* are active. Similar reasoning reveals that the remaining nodes are also visible. Note that we have *bound contextual variability* in the model by stating whether each context is active or not and by producing the corresponding version of the model. This process does not remove *non-contextual* variability from the model as shown in Figure 4-8A where two choices for applying the shipping discount remain. The selection among them can be made using the conventional goal model analysis techniques (e.g., [SGM04]).

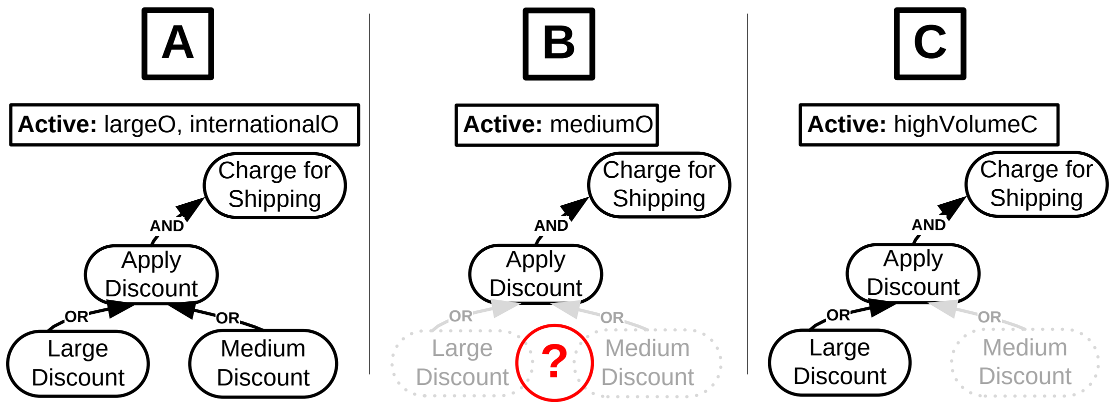


Figure 4-8. Analyzing the effects of domain variability on Charge for Shipping.

Figure 4-8B shows the model in the context of a medium order. Here, Charge for Shipping is visible again, as is Apply Discount since medium orders are substantial orders. However, there are no combinations of active tags (see Figure 4-7B for all the tag combinations implied in the model) that make the other two goals visible. The analysis reveals a problem with the resulting model since no refinement of a *non-leaf* goal Apply Discount is available and thus any goal depending on it will not be achieved. To mitigate this problem, one solution is to tag Medium Discount with *{{internationalO},{mediumO}}* instead of *{{internationalO}}*. Finally, Figure 4-8C shows the model resulting when the context *highVolumeC* is active. Since high-volume customers are important customers, they are given large discounts. The possibility of applying a medium discount is removed from the model.

## 4.6 Discussion

The social environment of the system is a great source of contextual information. Properties of various stakeholders – customers, supplier, regulators, etc. – usually affect the requirements of the system and changes in the social landscape of the system will most frequently demand a change in the system's behaviour. The simple context model presented in Figure 4-2A helps with keeping track of context entities and context dimensions but lacks the capabilities of advanced notations targeted at modeling socio-technical systems such as *i** or Tropos. The *i**'s high-level

Strategic Dependency (SD) models can be used to represent the social environment for the system, which, of course, is very useful in understanding who influences system requirements and how. Figure 4-9 shows the SD model centering on the Supply Customer BP actor that represents the system-to-be, the Supply Customer business process, and its relationships with other actors in its environment such as the Distributor, the Customer, the Supplier, and so on.

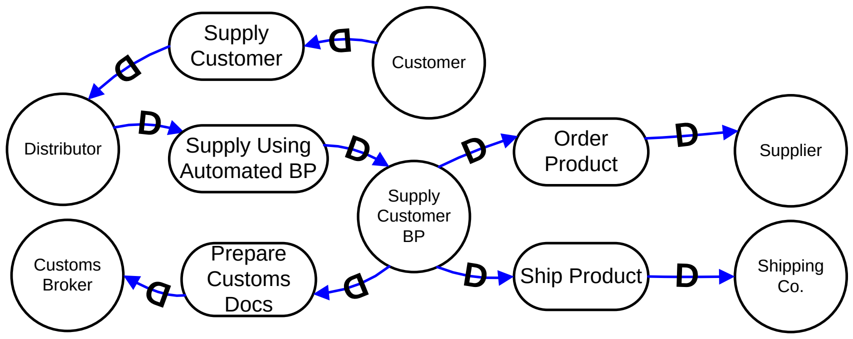


Figure 4-9. SD Model for the Supply Customer case study.

Here, the *actors*/stakeholders are represented by circles and their *intentional dependencies* are represented by directed arrows with the subjects of the dependencies (the goal nodes) in the middle. In this figure, only the some of the dependencies are represented. Obviously, there are non-functional (or softgoals) dependencies present as well (e.g., the Shipping Co. will be required to provide *dependable* deliveries). Since we are looking at the Supply Customer BP actor, the others can be viewed as context entities whose properties may influence the requirements. SD models can be used to help with the identification of actors in the system's environment. These actors become context entities in our approach if they affect system requirements. Note that in order for an actor to be considered a context entity it is not necessary for it to be related to the system through intentional dependencies. For instance, two actors can be competing for the services of another actor, thus having effect on each other without having any direct dependencies.

Our formal framework presented in Section 4.4 only deals with the visibility of context-dependent model elements. It does not guarantee that the resulting model is well-formed (e.g., as in Figure 4-8B). So, we need additional formalization for each modeling notation to construct and verify model variants given the sets of elements visible in specific contexts. Thus, our framework represents the generic component for reasoning about contextual variability upon which complete solutions can be built. An example of such a solution is our approach to context-enriched goal models, where, unlike in most goal-based RE methods, we always do goal refinement *in context*.

The hierarchical nature of goal models helped us to reduce the number of tags and to simplify the creation of context-enriched models. Other modeling notations can also benefit from the same idea. We have dealt with limited non-monotonic inheritance and are also exploring ways of modeling richer notion of context inheritance.

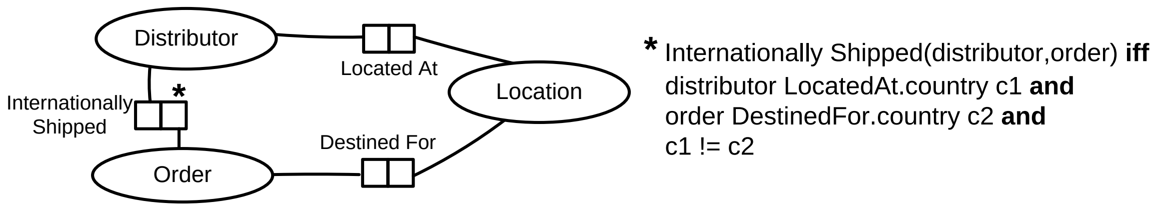

Distributor
Located At
Location
Internationally Shipped
*
Destined For
Order
* Internationally Shipped(distributor,order) **iff**
distributor LocatedAt.country c1 **and**
order DestinedFor.country c2 **and**
c1 != c2

Figure 4-10. The CML notation showing defined context *Internationally Located*.

In the approach presented in this chapter of the thesis, we do not capture relationships among contexts other than inheritance. In future work, we would like to be able to recognize which contexts are compatible and which are in conflict, to handle different contexts with different priorities and in general to be able to choose whether and under what circumstances to recognize the effects of contexts on requirements. We are looking into developing or adopting richer context modeling notations to help in analyzing and documenting domain variability in RE. In addition, we acknowledge that it is desirable to further analyze our formal notation to prove properties about it, especially in the presence of multiple and non-monotonic inheritance.

When determining the visibility of model elements (see *visible(n)* in Section 4.4.2), we require that either appropriate contextual tags be active themselves or that there be their active non-abnormal descendents. In our context framework, we leave it up to the modeler to make sure that context definitions are consistent with the context inheritance hierarchy specified for a particular model (i.e., that $ancestor(anc, dec) \supset active(dec) \supset active(anc)$).

The choice of the simple formal framework presented in Section 4.4 is based on the fact that it is intended to be used by designers and domain experts, so its simplicity was of the paramount concern. Other, more advanced notations, such as Description Logics (DLs) [BCMN03], could have been adapted to represent context hierarchies in our approach. DLs are based on a solid formal foundation and enjoy excellent tool support, but are quite complex for our context framework. In the future, as we strive to use the context framework for adaptive systems, we may re-evaluate the formal framework and choose a more expressive notation if the need arises.

The idea of context-dependent requirements models allows us to support a categorization of software requirements into the ones that are global, i.e., these are applicable regardless of the current context, and local – these requirements are only relevant (or become irrelevant) in certain contexts.

Specialized notations such as CML can be helpful for capturing complex contexts including derived ones. For example, in Figure 4-10, we are using the CML notation to model the context *Internationally Shipped*, which indicates that the destination for the Order and the Distributor are in different countries. This context is derived from two other contexts, *Located At* and *Destined For*, that correspond to the distributor location and order destination context dimensions in Figure 4-2A respectively. Defined[1] contexts provide a way for defining high-level application-specific contexts from simpler ones.

Context-awareness can be considered a type of adaptive behaviour where systems can both sense and react to changes in their environment. As software systems become more ubiquitous and as there are growing expectations of them being able to correctly adjust to changing environment conditions while achieving their objectives, the problem of systematically designing context-aware systems is becoming more and more important. The context framework presented in this thesis can serve as the foundation for the goal-driven approach for the elicitation and analysis of requirements for context-aware systems. It allows explicit modeling of the effects of various contexts on system requirements. However, context awareness is just one source of adaptive behaviour. Others include awareness requirements (requirements about requirements), which are discussed in Chapter 8 of this thesis, as well as maintenance goals. Along with the awareness-requirements approach presented in Chapter 8, these comprise some of the most important

[1] In CML, these are called *derived* contexts. However, we use the term *defined* since “derived” is normally used in inheritance.

components for the overall approach for the elicitation, modeling, and analysis of adaptation requirements.

The context framework can be used not only to capture the *external context* of the system (i.e., things that are outside of the system, but have an effect on it), but also the *internal context*. Internal context consists of things within the system that have important effects on the system behaviour. Internal faults and failures are just one example of what could be considered internal context. In Chapter 8, we use internal contexts to capture and refine the goals of feedback controllers that arise in the presence of various errors and failures in the system. Failure context hierarchies can be created and the handling of these failures (from the most general to the most detailed) can be explicitly modeled from the intentional point of view using goal models. Another example of a useful internal context is the performance of the system with respect to quality constraints. For instance, if the system is falling behind in terms of the time it takes to fulfill a customer order, it may need to acquire recovery goals designed to either make up the time or to somehow compensate the customer. The definitions of these contexts will, of course, rely on the presence of an infrastructure capable of monitoring not only the success/failure of goal achievement (here, we can employ the approach presented in [WMYM09]), but also of determining whether quality constraints are met. We plan to further explore this in the future.

The problem of the identification and management of conflicts among multiple viewpoints has attracted attention in the requirements engineering literature (e.g., [Eas93]). We have not dealt with inconsistencies and conflicts in our formal framework. The underlying assumption is that model elements tagged by the same combination of contextual tags constitute a consistent submodel of the original model. However, it is quite possible that a number of active contexts may have conflicting effect on the target model (e.g., on a goal model). For example, if the context *largeOrder* demands a heavy discount on the shipping price while the context *bulkyProduct* requires additional charges for shipping, we have to resolve the resulting conflict and produce a consistent goal model for this context. One possible solution is to prioritize contexts. Conflict detection and resolution remain part of the future work.

## 4.7 Conclusion

In this chapter, we have discussed a method for representing and reasoning about the effects of domain variability on goal models as well as the underlying generic framework for reasoning about visibility of context-dependent model elements. We use a well-understood goal modeling notation enriched with contexts to capture and explore *all* the effects of domain variability on requirements in a *single* model. Given a particular domain state, a goal model variation representing the requirements for that particular domain variation can be generated. We propose the use of context refinement hierarchies, which help in structuring the domain, in decoupling context definitions from their effects, and in incremental development of context-enriched goal models.

Taking domain variability into consideration allows us, in conjunction with the approach of [LLYY06], to increase the precision and usefulness of goal models, by explicitly capturing domain assumptions and their effects on software requirements.

# Chapter 5
# From High-Variability Goal Models to High-Variability Designs

**Acknowledgement**: This chapter is based on [YLLM08b]. We acknowledge the contribution of Yijun Yu to Sections 5.3.1, 5.3.2, and 5.3.3 and his major contribution to the development of tool support discussed here. We also acknowledge the contribution of Sotirios Liaskos to Section 5.3.3.

## 5.1 Introduction

Intentional variability in goal models refers to the fact that there are many different ways that stakeholder goals can be achieved. Intentional variability is the variability in the problem domain. In [LLYY06], Liaskos et al. propose a technique for the systematic elicitation and analysis of intentional variability in goal models. The elicitation, modeling, and analysis of this variability is beneficial for the requirements engineering process as it allows to explore the alternative ways for attaining high-level stakeholder goals and to select the best such alternative based on a number of criteria, most often with respect to non-functional requirements represented by softgoals.

In most goal-oriented requirements engineering approaches (e.g., [BPGM04], [Yu97]) intentional variability is bound at the level of requirements, i.e., the choices in the variation points are made before the requirements specification is produced. This process is aimed at selecting the best alternative for achieving system goals given the desired preferences, priorities, etc. The benefit of this is that system design and implementation will only deal with the selected alternative, which was considered the best available at the time the analysis was done. This, of course, allows the software development organization to minimize the use of resources for the design and implementation of the target system. However, binding intentional variability at this stage also has its drawbacks. Here are some of the most important ones. First, while the selection of the best alternative may be a result of well-thought out and well-executed process, due to the dynamic nature of most application and business domains, complex competitive pressures on organizations, shifting user preferences, etc. it is possible and quite probable that at a later point in time the selection will appear suboptimal, if not wrong. Second, due to their diversity, often it is not possible to select a single system configuration for all of the existing or potential users of the system. Therefore, the software development effort usually targets the solution that works for the majority of users/customers of the system while ignoring the ones that are considered non-standard (e.g., users with disabilities). This leads to a certain proportion of potential users of the system not being accommodated by the software, which may lead to lost revenue as well as to possible legal challenges. Finally, in case the system is unable to achieve its goals using the implemented "best" alternative, due to the lack of other means for attaining its goals (even if they are suboptimal), it will not have the capacity to be easily reconfigured to circumvent the failure.

To avoid the above difficulties, we propose that despite some pruning of alternatives in the problem domain based, for example, on their feasibility or their fitness with respect to the desired quality constraints, the remaining problem domain variability captured in goal models must be preserved in the solution domain – throughout the subsequent stages of the software development process. This way, the implemented system will have a number of alternative ways to achieve its goals (i.e., alternative behaviours). While all of these behaviours deliver the same functionality, they are usually different in terms of their non-functional properties and possibly other characteristics. A software system implemented in this manner is armed with a number of alternatives for attaining its goals, which opens up a possibility for selecting particular behaviours based on the desired customization/configuration (at design time, instantiation time, or at runtime). Alternatively, a multitude of implemented behaviours provide the basis for dynamic selection of behaviours at runtime, i.e., for the development of adaptive systems. Thus, preservation of requirements-level variability in the solution domain is the foundation for our approach for designing customizable/configurable or adaptive/adaptable software systems.

We call *generic* a software system that accommodates many different behaviours, all delivering the same function with possibly varying non-functional properties. Such systems exhibit *variability* in the way they achieve their objectives, i.e., in the solution domain. The selection of the particular behaviour can be based on several criteria, for example, on the evaluation of the alternative behaviours with respect to non-functional criteria, preferences/priorities or other constraints. In this thesis chapter, we aim at using goal models to develop such generic software solutions. The approach is requirements-driven and involves the elicitation of variability in the problem domain and its capture using HV goal models together with variability-preserving transformations to various solution-oriented models. These target models usually require the identification and modeling of the details of the problem domain that cannot be captured by the standard goal modeling notation. For example, to generate statecharts representing the system behaviour, it is crucial to capture the dependencies among the goals in the source goal model. We need to determine whether or not goals can be achieved independently of each other (the parallel composition of goals), or whether some depend on others (the sequential composition), etc. These goal dependencies are properties of the problem domain, so it is natural to capture them at the level of requirements using goal models. Thus, to model these additional details we augment the standard goal models with *annotations*. The type of annotations we use depends on the target modeling notations, which may require the modeling of different aspects of the problem domain. The idea of goal model annotations is not new. For example, it was widely used by Lapouchnian and Lespérance in [LL06] for augmenting high-level *i** models for their subsequent transformation into formal agent specifications.

For mapping goal models into a variety of notations we identify a small number of mappings linking goal model patterns to the corresponding patterns in the target notations. Using the hierarchical nature of goal models these generated model fragments are then combined to produce a complete target model. This approach works well with hierarchical target models. However, in Chapter 6 we discuss the mapping of HV goal models into a non-hierarchical modeling notation.

One of the key ideas in these transformations is that they not only preserve requirements-level variability in the solution domain, but the choices among the alternatives in the generated models are linked back to the selections within the appropriate goal model variation points. This traceability allows us to use goal models as high-level abstractions of the systems when dealing with the problems of their configuration or behaviour selection. Then, goal model reasoning techniques [SGM04], [GMNS02] can be employed for reasoning about system configurations.

In this chapter, we present a number of variability-preserving transformations illustrating the above ideas. We target three complementary design views: a feature model, a statechart and an architectural component-connector model. The feature model describes the system-to-be as a combination of variable sets of features. This notation is extensively used in product line engineering to select particular product configurations out of a product family. Statecharts provide a view of the behaviour alternatives in systems, while component models reveal the view of alternatives as variable structural bindings of software components.

The source goal model is used as the logical view at the requirements stage, somewhat similar to the global view in the 4+1 views [Kru95] of the Rational Unified Process. This goal model transcends and circumscribes design views.

## 5.2 From Goal Models to High-Variability Designs

We now outline the generic process that is used for requirements-driven and variability-preserving generation of solution-oriented models from goal models. Here are the main steps of the process. These steps are to be customized for the particular target notation.

1. Create a high-variability goal model. The emphasis here is on capturing all the possible ways to attain high-level goals. Approaches such as [LLYY06] can be used to help with the systematic identification of such variability. Intentional OR decompositions in the model are labelled as variation points.

2. Based on the characteristics of the target modeling notation, identify problem domain aspects, which need to be captured at the requirements level for the subsequent translation into the target notation, but cannot be modeled using the standard goal modeling notation. Use a minimal set of goal model annotations to represent the required information. We stress that this information is part of the problem, not the solution domain. Thus, care must be taken not to capture the information that may be required by the target notation, but that is too low-level or too solution-oriented to be dealt with at the requirements level. That information will have to be manually added later to the generated model.

3. Map the HV goal model into the target model while preserving the variability.

    a. Identify the mapping of leaf-level goals into the target notation. For most solution-oriented modeling notations, leaf-level goals will be mapped into concrete units of behaviour/structure (e.g., operations, activities, method/service invocations, components, etc.)

b. Determine whether softgoals and softgoal contributions can be mapped into the target notation. In most cases, these will not have counterparts in the solution-oriented modeling notations. They still can be used for reasoning about the behaviour or the configuration of the target system due to the traceability between the system and the goal model that is created after the mapping process.

c. Identify how goal decompositions with annotations will be mapped into the target notation. In many cases, these will be mapped into compositions of lower-level model fragments. For instance, sequential or parallel compositions can be used to join business process activities that result from the mapping of leaf-level goals (see Chapter 6). Care must be taken when mapping variation points. All of the choices in a VP must be represented in the target notation. Additionally, the alternatives corresponding to the VP choices in the source goal model need to refer to those choices through the appropriate parameterization. This way, the selections made in the goal model using goal reasoning algorithms or other means can be reflected in the target models.

d. Using the hierarchical nature of goal models combine the model fragments resulting from the mapping of leaf-level goals using the mapping patterns for goal refinements. This step is normally automated.

4. The result of the previous steps of the process may be a *preliminary* model in the target notation. Thus, we need to provide additional details required by that notation, but missing from the automatically generated high-variability model. At the end of this process step a complete model in the target notation is produced. This model preserves the variability in the source goal model while variation point choices in the generated model refer to the selections made in the corresponding VPs in the goal model.

## 5.3 Generating Preliminary Design Views

In this section, we apply the generic process for variability-preserving generation of design-level models from high-variability models presented in the previous section to three modeling notations. A design parameterized with the variation point choices is a *generic* design. Such design can implement, for example, a product-line, an adaptive system or a flexible component-based architecture, among other things. A *product-line* can be configured into various products that share/reuse the implementation of their commonality. An *adaptive* product can adapt its behaviour at run-time to react to or accommodate anticipated changes in the environment through the use of alternative solutions. A *component-based* architectural design makes use of interface-binding to flexibly orchestrate components to fulfill the goal. In this section, we discuss the three target solution-oriented views: feature models, statecharts, and component-connector diagrams. Intentional variability present in the source goal models is preserved as *configuration variability* in feature models, *behavioural variability* in statecharts, and *structural variability* in connector-component models. In this work, traceability between requirements and design is maintained by an explicit transformation from goal models to design views as well as variation point parameterization.

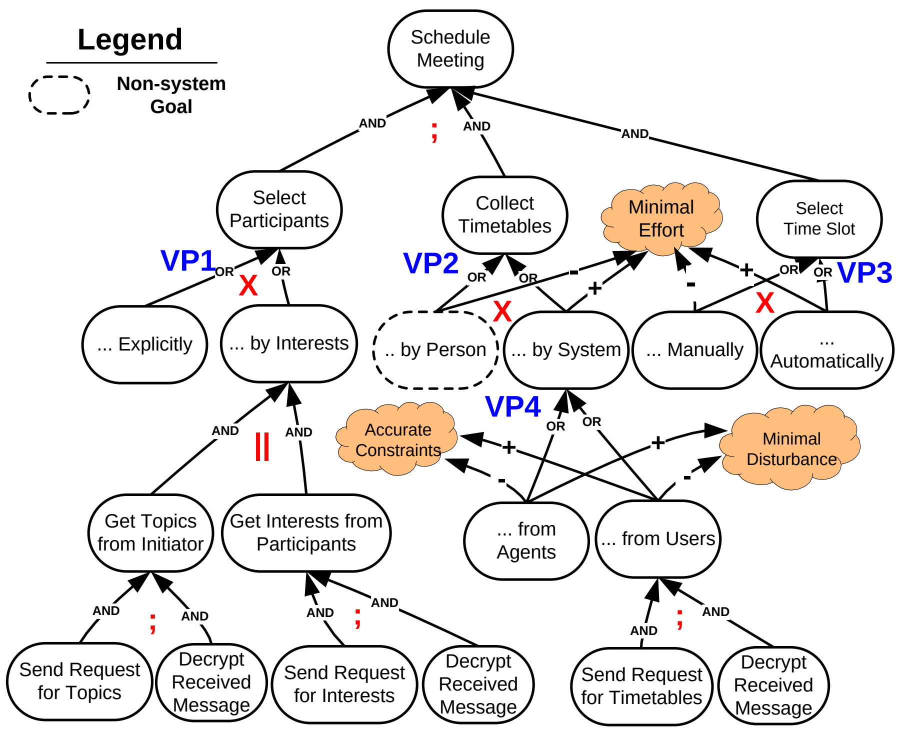


Figure 5-1. Goal model annotations.

Figure 5-1 presents an annotated goal model showing the refinement of the goal Schedule Meeting that will be used in the subsequent sections of this chapter for feature model, statechart, and connector-component model generation. It is based on the goal model in Figure 3-1. Four variation points labelled "VP1" through "VP4" are identified. For AND decompositions we identify whether the subgoals are to be achieved in parallel (||) or sequentially (;). For OR decompositions we specify whether the OR refinement is exclusive (X). For example, the OR decomposition of the goal Select Time Slot (which is the variation point VP3) is exclusive since the time slot can either be selected manually or automatically. Similarly, the sequential annotation indicates that the subgoals of the goal Get Topics from Initiator are to be achieved in order from left to right. Also, goals that are to be delegated to non-system agents are indicated by dashed borders (e.g., the goal Collect Timetables by Person). This annotation means that the responsibility for achieving the goal does not lie with the system. Therefore, the achievement of the goal will not be part of the system behaviour. We will explain the detailed annotations for deriving a component-connector view in the next section. As the preliminary design evolves, design elements and their relationships can change dramatically. However, the traceability to the required variability must be maintained so that one can navigate between the generated design views using the derived traceability links among them.

### 5.3.1 Generating Feature Models

As mentioned in Section 2.3, feature modeling is a domain analysis technique that is targeted at modeling commonality and variability among products in product lines. A feature model is a compact representation of the different product variations that belong to a product line. A *feature* is usually an increment in program functionality. There are four main types of features in feature modeling: *Mandatory*, *Optional*, *Alternative*, and *OR* features [CE00]. A Mandatory feature must be included in every member of a product line family provided that its parent feature is included; an Optional feature may be included if its parent is included; exactly one feature from a set of Alternative features must be included if a parent of the set is included; any non-empty subset of an OR-feature set can be included if a parent feature is included.

There are fundamental differences between goals and features. Goals represent stakeholder intentions (which are manifestations of intent) that may or may not be realized. Thus, goal models represent a space of intentions, which may or may not be fulfilled. Features, on the other hand, represent properties of concepts or artefacts [CE00]. Goals will use the services of the system-to-be as well as those of external actors for their fulfillment. Features in product families represent *system* functions or properties. Goals may be partially fulfilled in a qualitative or quantitative sense [GMNS02], while features are either elements of an allowable configuration or they are not. Goals may come with a modality: achieve, maintain, avoid, and cease [DLF93], while features have none. Likewise, AND decomposition of goals may introduce temporal constraints (e.g., fulfill subgoal A before subgoal B) while feature refinements do not.

As noted in [CE00], feature models must include the semantic description and the rationale for each feature (i.e., why it is in the model). Also, variable (OR/Optional/Alternative) features should be annotated with conditions describing when to select them. Since goal models already capture the rationale (stakeholder goals) and the quality criteria driving the selection of alternatives, we hypothesize that they are the proper candidates for the generation of feature models.

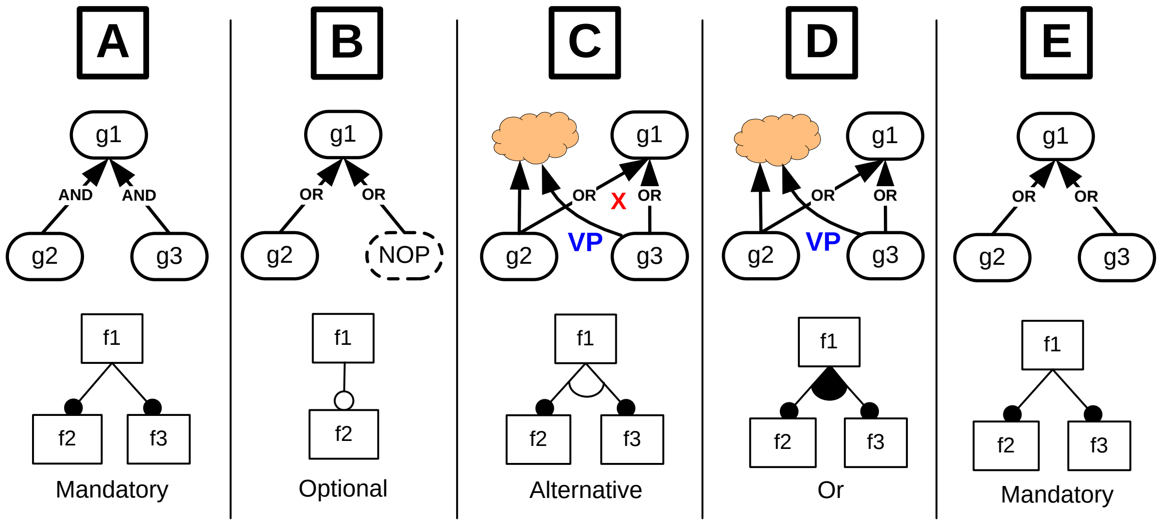


Figure 5-2. A set of patterns for generating feature models.

We now discuss the process of generating initial feature models from HV goal models. Since a feature model represents the variability in the system-to-be, in order to generate it we need to identify the subset of a goal model that is intended for the system under development. This information is not part of the standard goal modeling notations. Therefore, as previously discussed, appropriate annotations need to be applied to goal models in order to generate the corresponding preliminary feature models. AND decompositions of goals generally correspond to sets of Mandatory features (see Figure 5-2A). For OR decompositions, it is important to distinguish two types of variability in goal models: *intentional* (or design-time) and *runtime*. Intentional variability is high-level and it is independent of input. It can usually be bound with the help of user preferences or quality criteria. In our models, intentional variability is labelled as variation points ("VP"). For feature model generation, variation points by default are mapped into sets of OR features (Figure 5-2D). Alternative features do not have an immediate counterpart in goal models. However, with the help of exclusive OR annotations applied to VPs (there, exactly one choice must be selected), we can generate alternative feature sets (Figure 5-2C). On the other hand, runtime, dynamic variability depends on runtime input and thus must be preserved in the system. This type of variability is usually event or data-driven. For example, participants for a meeting can be selected explicitly (by name) or chosen by matching the topic of the meeting with their interests (see Figure 5-1). The choice will dynamically depend on the meeting type, which is to be selected at runtime for every instance of the meeting scheduling process. Thus, all of the alternatives must be implemented and available for selection at runtime. So, runtime OR decompositions will map into mandatory feature sets (Figure 5-2E). Finally, when a goal is OR-decomposed into at least one non-system subgoal (specified by a goal annotation NOP or the dashed goal node in the model), the sibling system subgoals will be mapped into optional features (Figure 5-2B). Non-system (NOP) goals are not mapped to any feature in the feature model. In a more realistic design, however, when dealing with NOP goals the system may need to facilitate the environmental actors in achieving their goals or monitor the achievement of these goals. Here, the goals delegated to the environment can be replaced with user interfaces, monitoring or other appropriate features. In general, there is no one-to-one correspondence between goals delegated to the system and features. While high-level goals may be mapped directly into grouping features in an initial feature model, a leaf-level goal may be mapped into a single feature or multiple features, and several leaf goals may be mapped into a feature by means of factoring. For example, a number of goals requiring decryption of received messages in a secure meeting scheduling system may be mapped into the single feature "Message Decrypter" (see Figure 5-4[2]).

[2]One can systematically derive feature names from the hard goal descriptions by, for instance, changing the action verb into the corresponding noun (e.g., "Schedule Meeting" becomes "Meeting Scheduler").

In addition to generating feature models themselves, goal models can help with the identification of feature model constraints. *Constraints* in feature models represent relationships among variable features that cannot be captured by feature decompositions [CE00]. These constraints include, for example, mutual exclusion and mutual dependency. To help in feature selection, given a goal model, feature model constraints can be generated linking features that have their corresponding goals contributing (positively or negatively) to the same softgoal. For instance, if two system-delegated goals contribute positively (respectively, negatively) to the softgoal S, then both their corresponding features will most likely have to be included in (respectively, excluded from) the system provided that the softgoal is of importance for that system variant. Thus, we generate a mutual dependency constraint between the two features. The constraint's label includes the strength of the softgoal contribution and the name of the softgoal to document the source of the constraint (e.g., `+depends[S]`, if both goals contributed positively to S). Similarly, if two system-delegated goals have the opposite contributions to a softgoal, then selecting both corresponding features in a system that tries to satisfice the softgoal will be counterproductive. This will result in a mutual exclusion constraint (e.g., `+conflicts[S]`) between the two features. Thus, the constraints help in the feature selection process by accounting for stakeholders' quality concerns.

In general, to obtain a feature model constraint between two features $f_X$ and $f_Y$ based on the softgoal contributions of their corresponding goals, X and Y, to the softgoal S we use the following rules, see Figure 5-3. Here, `+(X,S)` indicates that the goal X contributes positively to the softgoal S, `-(X,S)` indicates that the goal X contributes negatively to the softgoal S, etc.

```
  +conflicts[S](X, Y) □  (+(X,S) AND -(Y,S) OR -(X,S) AND +(Y,S))
++conflicts[S](X, Y) □  (++(X,S) AND --(Y,S) OR --(X,S) AND ++(Y,S))
  +depends[S](X,Y) □  (+(X,S) AND +(Y,S))
  -depends[S](X,Y)  □  (-(X,S) AND -(Y,S))
++depends[S](X,Y) □  (++(X,S) AND ++(Y,S))
- - depends[S](X,Y) □  (--(X,S) AND --(Y,S))
```

Figure 5-3. Feature model constraint generation rules.

The constraints are parameterized by the softgoal S to indicate that they are significant only when S is important to stakeholders. Similarly, the strength of softgoal contributions (involving the goals that led to the dependent or conflicting features) implies the strength of the constraints (e.g., +|–|++|– –).

The result of the transformation of a goal model shown in Figure 5-1 into a feature model is presented in Figure 5-4. The implemented transformation procedure has been reported in [YLLM08a].

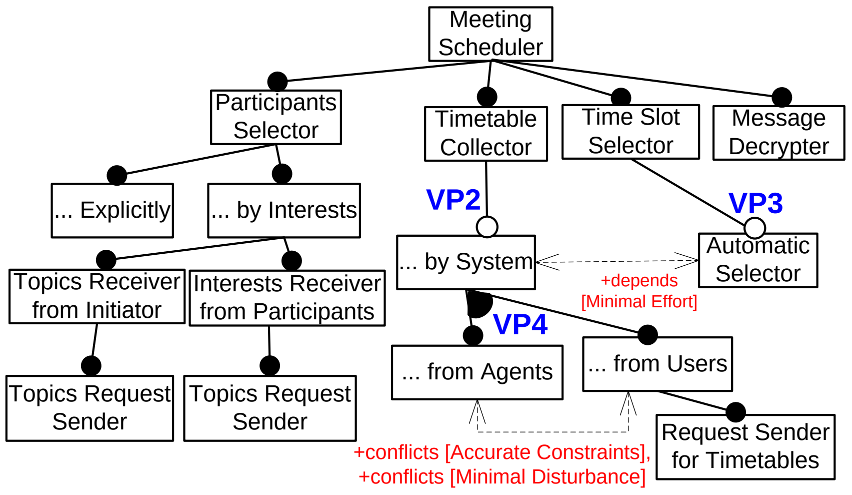


Figure 5-4. Feature model generated from Figure 5-1.

The generated feature models reflect the fact that decompositions in goal models are much more restrictive than in feature models where feature types can be mixed within a single decomposition. Thus, we produce feature models where features have sub-features of a single type and cannot have more than one set of Alternative or OR features. One can further group them into mixed-type feature decompositions if appropriate.

There are additional feature types defined by Czarnecki and Eisenecker [CE00] based on their mapping into software components, such as *concrete* features that may be realized as separate components, *aspectual* features that affect a number of components, *abstract* features such as performance, and so on, and *grouping* features that may either correspond to a common component interface or be used for organizational purposes. In our approach, we have not addressed the problem of recognizing these types of features in the generated feature models since they are at the level of architectural design, not requirements. However, it is clear that certain elements in our goal models may play a role in identifying these feature types. For example, non-functional requirements are a common source of *aspects* in software development.

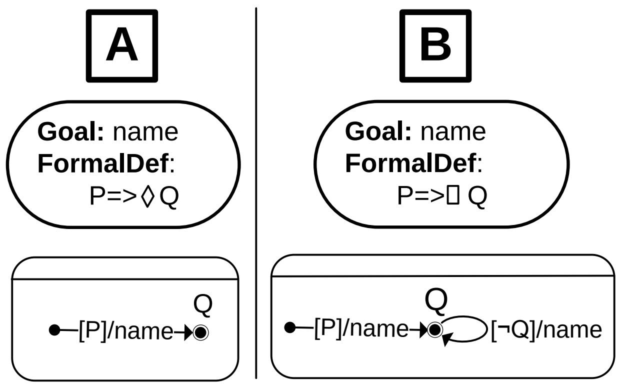


Figure 5-5. Statechart patterns for *achieve* (A) and *maintain* (B) goals.

## 5.3.2 Generating Statecharts

Statecharts, as proposed by David Harel [HN96], are a visual formalism for describing the behaviour of complex systems. On top of states and transitions of a state machine, a statechart introduces a nested super-/sub-state structure for abstraction (from a state to its super-state) or decomposition (from a state to its substates). In addition, a state can be decomposed into a set of *AND* states (visually separated by swim-lanes) or a set of *XOR* states [HN96]. A transition can also be decomposed into transitions among the substates. This hierarchical notation allows the description of a system's behaviour at different levels of abstraction. This property of statecharts makes them much more concise and usable than plain state machines. Thus, they constitute a popular choice for representing the behavioural view of a system. Figure 5-5 shows a pair of mappings from goals in a goal model to states in a statechart. In general, one can identify four basic goal patterns: *achieve*, *maintain*, *cease*, and *avoid* [DLF93]. An *achieve* leaf-level goal is expressed in Figure 5-5A as a temporal formula with *P* being its precondition, and *Q* being its post-condition. In the corresponding statechart, one entry state and one exit state are created: *P* describes the condition triggering the transition from the entry to the exit state; *Q* prescribes the condition that must be satisfied at the exit state. The transition (named "name", the same name as the goal's) is associated with an activity to reach the goal's desired state. The *cease* goal is mapped to a similar statechart (not shown) by replacing the condition at the exit state with $\neg Q$. Figure 5-5B shows the mapping of a *maintain* leaf-level goal into a statechart pattern. Here, we have an additional transition that restores the state back to the one that satisfies *Q* whenever *Q* is violated while *P* is satisfied. Similar to the *maintain* goal's statechart, the statechart for an *avoid* goal swaps *Q* with its negation. These conditions can be used symbolically to generate an initial statechart view, i.e., they do not need to be explicit temporal logic predicates. At the detailed design stage, the designer may provide solution-specific information to specify the predicates for simulation or execution of the refined statechart model.

Then, by applying the patterns in Figure 5-6 (A-D), a goal hierarchy will be mapped into an isomorphic state hierarchy in a statechart. That is, the state corresponding to a goal becomes a super-state of the states associated with its subgoals. The runtime variability will be preserved in the statecharts through alternative transition paths.

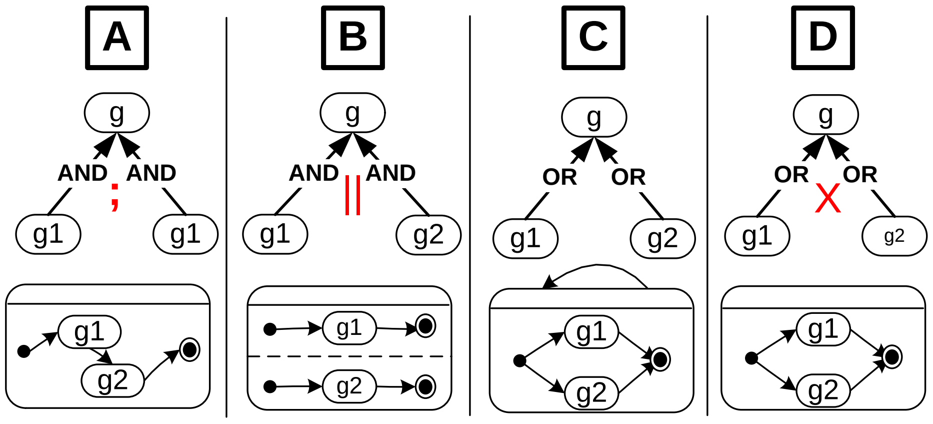


Figure 5-6. Mapping goal decompositions to statechart patterns.

The transformation from a goal model to an initial statechart can be automated even when the temporal formulae are not given: we first associate each leaf goal with a state that contains an entry substate and an exit substate. A default transition from the entry substate to the exit substate is labelled with the action corresponding to the leaf goal (Figure 5-5). Then, the AND/OR goal decompositions are mapped into compositions of the statechart fragments. In order to know how to connect the substates generated from the corresponding AND-decomposed subgoals, control flow constraints, which are introduced as goal model annotations, are utilized. Figure 5-1 shows the Schedule Meeting goal model augmented with the necessary control flow annotations, namely parallel and sequential compositions of subgoals. The details of the model will be discussed later in the section. Note that the parallel AND decomposition is mapped into *AND* states with swim-lanes (Figure 5-6B). Additionally, for an OR-decomposition, one has to consider whether it is inclusive (Figure 5-6C, note the transition from the super-state to itself to support the execution of multiple alternatives) or exclusive (Figure 5-6D).

Given root goals, our statechart generation procedure descends along the goal refinement hierarchy recursively. For each leaf goal, a state is created according to Figure 5-5. The created state has an entry and an exit substate. Next, annotations that represent the temporal constraints on AND/OR goal decompositions are considered. Composition patterns (Figure 5-6 (A-D)) can then be used to combine the statecharts of subgoals into one statechart. In particular:

1. When a goal is AND-decomposed sequentially (;) into *N* subgoals (Figure 5-6A), we create *N* + 1 transitions that connect the *N* subgoal states with the entry and exit states of the goal as a sequential chain. The sequence of substates and transitions corresponds to the order of the subgoals (from left to right) in the goal model.

2. When a goal is AND-decomposed into *N* subgoals with the concurrency annotation (||) (Figure 5-6B), we create *N* pairs of transitions that connect each subgoal state with the entry and exit states of the goal. Then, the states are become the AND decompositions of the super-state in the statechart.

3. When a goal is OR-decomposed into *N* subgoals inclusively (Figure 5-6C), we create *N* pairs of transitions that connect each subgoal state with the entry and exit states of the goal respectively, and also create a cyclic transition for fulfilling any number of subgoals in the OR decomposition.

4. When a goal is OR-decomposed into *N* subgoals with an exclusive annotation (Figure 5-6D), we create *N* pairs of transitions that connect each subgoal state with the entry and exit states of the goal respectively. The states are the XOR decomposition of the super-state in the statechart.

The upper bound of number of states generated from the above patterns is *2N* where *N* is the number of goals.

One important addition to the above rules is the following. Whenever we are mapping a variation point *VPn* into a statechart, we will proceed with the mapping as we would for an OR goal decomposition (Figure 5-6D), but will add appropriate guard conditions to the transitions as indicated in Figure 5-7. The guard conditions are utilized to make sure that only the transitions corresponding to the selections made in the VP are enabled in the statechart.

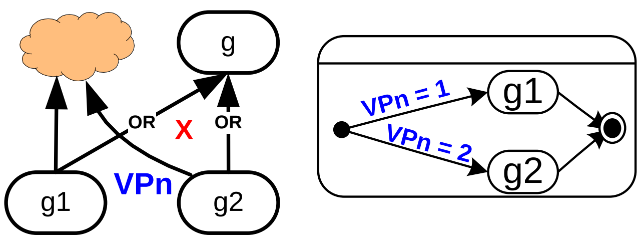


Figure 5-7. Mapping variation points.

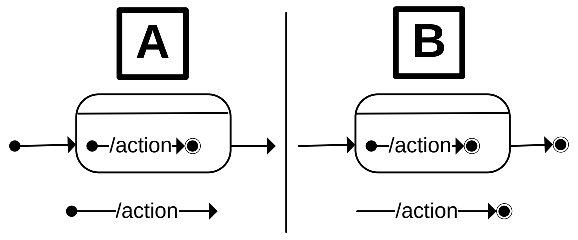


Figure 5-8. Simplification patterns for statecharts mapped from leaf-level goals.

The statechart generated using the previously discussed patterns can be simplified when the goals are at the leaf level (Figure 5-8): since no new intermediate state is introduced between the entry and the exit state, the action on a single transition will be moved to an incoming transition from the sibling entry super-state (Figure 5-8A) or to an outgoing transition to the sibling exit super-state (Figure 5-8B).

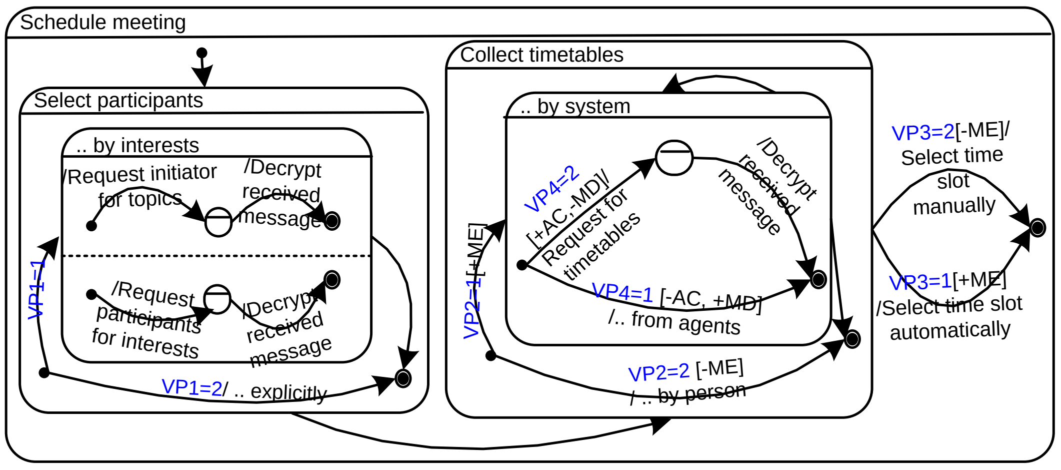

Schedule meeting
Select participants
.. by interests
/Request initiator for topics
/Decrypt received message
/Request participants for interests
/Decrypt received message
VP1=1
VP1=2/ .. explicitly
Collect timetables
.. by system
VP4=2 [+AC,-MD]/ Request for timetables
/Decrypt received message
VP4=1 [-AC, +MD] /.. from agents
VP2=1[+ME]
VP2=2 [-ME] / .. by person
VP3=2[-ME]/ Select time slot manually
VP3=1[+ME] /Select time slot automatically

Figure 5-9. Statechart generated from Figure 5-1.

To produce the corresponding statechart given the original Schedule Meeting goal model in Figure 3-1, we need to capture the problem domain aspects that the standard goal model in Figure 3-1 is unable to capture. We use the appropriate annotations to produce the enriched version of the model presented in Figure 5-1. Here, we first identify the sequential/parallel control patterns for AND decompositions through the analysis of the data/control flow dependencies. For example, there is a data dependency from Send Request for Timetable to Decrypt Received Message because the timetable needs to be requested first, then received, and then decrypted. Second, we identify the inclusive/exclusive patterns for the OR decompositions. For example, Select Time Slot is done either Manually or Automatically. Then we generate the statechart according to the patterns in Figure 5-5 and Figure 5-6. As a result, we obtain a statechart with the hierarchical state decompositions (see Figure 5-9). It describes an initial behavioural view of the system.

The preliminary statechart can be further modified by the designer. For example, the abstract “request for timetables” state can be further decomposed into a set of substates such as “send individual request for timetable” for each of the participants. Since goal variability in goal models is preserved through the behavioural variability in the statecharts and the correspondence between variation points in these two notations is established by the guard conditions on the transitions, changes to statecharts can still be traced back to the corresponding annotated goal models.

### 5.3.3 Generating Component-Connector Diagrams

An important software engineering principle is to modularize a system into a set of subsystems (i.e., modules, components) that have low coupling and high cohesion [Par72]. A component-connector architectural view is typically represented using an architecture description language (ADL) [MT97]. We adapt the Darwin ADL [MK96] with elements borrowed from one of its extensions, namely Koala [OLKM00]. Our component-connector view is defined by components and their bindings through interface types. An *interface type* is a collection of message signatures by which a component can interact with its environment. A *component* can be connected to other components through instances of interface types (i.e., interfaces). A **provides** interface shows how the environment can access the component’s functionality, whereas a **requires** interface shows how the component can access the functionality provided by its environment. In a component configuration, a **requires** interface of a component in the system must be *bound* to exactly one **provides** interface of another component. However, as in [OLKM00], we allow alternative bindings of interfaces through the use of a special connection component, the *switch* (see Figure 5-10A). A switch allows the association of one **requires** interface with two or more alternative **provides** interfaces of the same type, and is placed within a compound component which contains the corresponding alternative components. The switch represents alternative bindings among interfaces. It is, therefore, the tool we use to support variability in architectural representations generated from goal models.

A preliminary component-connector view can be generated from a goal model by creating an interface type and a component for each goal. The interface type contains the signature of an operation. The operation name is directly derived from the goal description, the **IN/OUT** parameters of the operation signature must be specified for the input and output of the component. Therefore, in order to generate such a preliminary component-connector view, each goal needs to be annotated with specification of input/output data. These could take the form of parameters. We also use goal parameters in the design of business processes in Chapter 6. Here, inputs are the data that need to be provided to the agent responsible for the achievement of the goal in order to fulfill it. Outputs are the data that the agent provides to its environment as part of the delivery of the goal. For instance, the parameters "Initiator Address" and "Topics" are the respectively the input and the output of the goal Get Topics from Initiator.

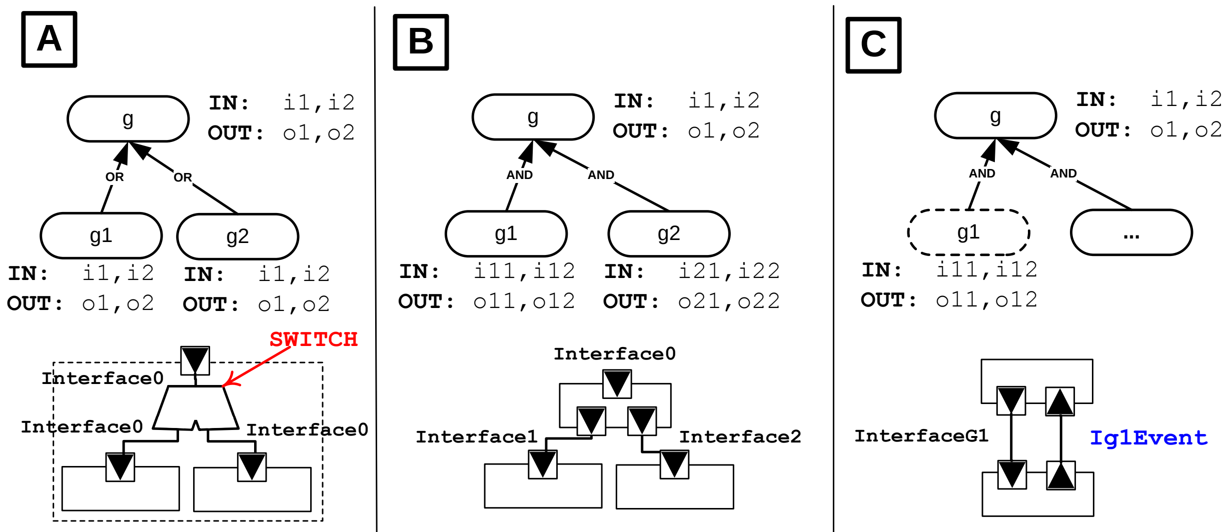
We produce the initial architectural model from a goal model in the following way. First, based on the specified inputs/outputs, we create an interface type and a component for each goal in the goal model. The associated interface type of a goal initially contains one operation signature, the name of which is directly derived from the description of the goal. The inputs and outputs of the goal become the IN and OUT parameters of the signature. For example, the goal Collect Timetables from Users with the inputs "Users" and "Interval" and the output "Constraints" produces the following interface:

```
interface type ICollectTimetablesFromUsers {
CollectTimetables(IN Users, Interval, OUT Constraints);
}
```

The generated component implements the interface type through a **provides** interface. The **requires** interfaces of the component, on the other hand, depend on how the goal is decomposed. If the goal is AND-decomposed (as in Figure 5-10B), the component will have as many **requires** interfaces as the subgoals. In our example, the initial architectural component for the goal Collect Timetables from Users is generated as follows:

```
component TimetableCollectorFromUsers {
        provides ICollectTimetables;
        requires IGetTimetable, IDecryptMessage;
}
```

The **requires** interfaces are bound to the appropriate **provides** interfaces of the subgoals. The name of the component is based on its **provides** interface.



```
interface type Interface0{
   G(IN i1, IN i2
   OUT o1, OUT o2);
}
```

```
interface type Interface0{
 G(IN i1, IN i2, OUT o1, OUT o2);
}
interface type Interface1 {
 G1(IN i11, IN i12, OUT o11, OUT o12);
}
interface type Interface2 {
 G2(IN i21, IN i22, OUT o21, OUT o22);
}
```

```
interface type InterfaceG1 {
 G1(IN i11, IN i12);
}

interface type Ig1Event {
 G1Event(OUT o11, OUT o12);
}
```

Figure 5-10. A set of goals-to-architecture patterns.

If the goal is OR-decomposed (Figure 5-10A), then the corresponding component in the architectural model becomes a compound one. Then, interface types of the subgoals are replaced with the interface type of the parent goal such that the **provides** interface of the parent goal is of the same type as the **provides** interfaces of the subgoals. Then, inside the generated component, a switch is introduced to bind these interfaces. The **provides** interface of the compound component generated for the parent goal can be bound to any of the subgoals' **provides** interfaces. Both the switch and the components of the subgoals are placed inside the component of the parent goal, and are *hidden* behind its interface.

In Figure 5-10 as well as in Figure 5-11, the graphical notation is directly adopted from Koala/Darwin. The boxes are components and the arrows attached to them represent **provides** and **requires** interfaces, depending on whether the arrow points inwards or outwards respectively. The lines show how interfaces are bound for the particular configuration and are annotated with the name of the respective interface type; the shape of the overlapping parallelograms represents a switch (as in Figure 5-10A). The corresponding goal model variation point is specified next to switch components.

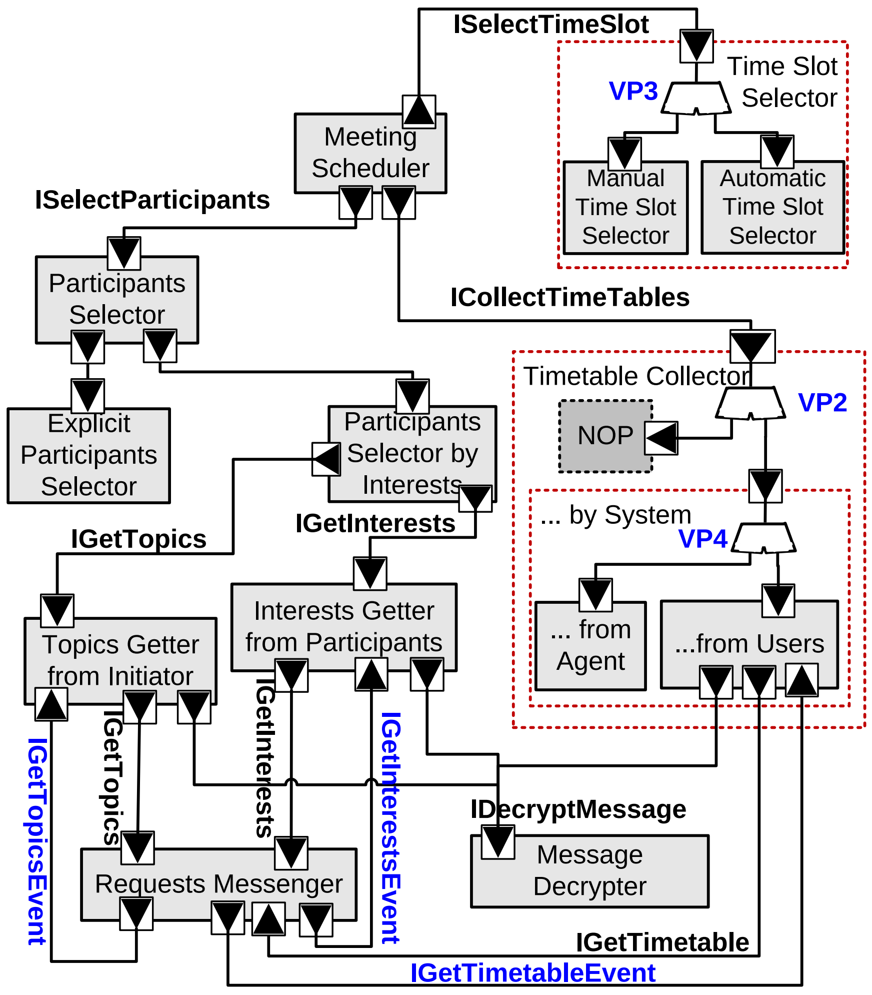


Figure 5-11. The architectural diagram generated from Figure 5-1.

In order to accommodate a scheme for *event* propagation among components, we follow an approach inspired by the C2 architectural style [MT97] where requests and notifications are propagated in opposite directions. As requests flow from high level components to low-level ones, notifications (events) originated from low-level components will propagate to high-level ones. Such events are generated from components associated with goals that are delegated to the environment (non-system goals). These components are responsible for supporting external actors' activity to attain the associated goal, for sensing the progress of this attainment and for communicating this to the components of the higher level by generating the appropriate events. We name such components *interface components* to signify that they lay at the border of the system. Interface components have an additional **requires** interface that channels the events (see Figure 5-10C for details of this mapping). This interface is optionally bound to an additional **provides** interface at the parent component, its event handler. A typical binding example is a Java `Listener` interface implemented in the parent component for receiving events from the interface component. The naming convention `I[GoalName]Event` is used here to distinguish event interfaces. In our Meeting Scheduler example (see Figure 5-11), three goals – Send Request for Topics, Send Request for Interests, and Send Request for Timetable – are delegated to external actors (e.g. meeting initiators and meeting participants), and will therefore each yield an interface component, e.g.:

```
component TimetableCollectorFromUsers {
        provides IGetTimetable, ITimetableEvent;
        requires IGetTimetable, IDecryptMessage;
}
```

The `Requests Messenger` component is a result of the merging of lower-level components and is being reused by three different higher-level components through parameterization. These are examples of modifications the designer may choose to make after the preliminary model has been produced.

## 5.4 Tool Support and Traceability

The OpenOME [OOME07] modeling tool is capable of editing goal models, adding annotations, and generating the design views such as feature models [YLLM08a] and BPEL processes (as in Chapter 6 of this thesis). A simplified goal model in Figure 5-1 is represented by the XML document in Figure 5-12A. OpenOME is an Eclipse-based goal graph editor, with the models stored in the XMI [XMI07] format. An annotated goal model is presented in Figure 5-12A. Here, the default namespace (goals) is declared with the use of the `xmlns` attribute. The attribute informs the goal graph editor that the model is a goal model. In the representation, next to the goals and their annotations, the sibling tags from different namespaces provide traceability links to other views. The attribute `src` of the elements in the generated solution-oriented models (Figure 5-12B) points to the original element from the goal model, the source of transformation. When the attribute has a value other than `generated`, it indicates that the design element has been manually changed. Thus, if the source goal model is changed, the renamed design element will not be overridden by the generation process.

**A** Annotated high-variability goal model

```
<view xmlns="urn:goals" xmlns:e="urn:enrichments"
  xmlns:f="urn:features" xmlns:s="urn:statecharts"
  xmlns:c="urn:components">
  < goal name="Schedule Meeting">
    <refinement type="goal" value="AND"/>
    <e:refinement type="state" value="SEQ"/>
    <e:annotation type="feature" value="SYS"/>
    <e:annotation type="component"
                  input="meeting" output="schedule"/>
    <e:annotation type="state" pre="meeting!=null"
             post="!schedulable OR schedule!=null"/>
    <f:feature name="Meeting Scheduler"
                        src="generated: renamed"/>
    <s:statechart name="Schedule Meeting"
                                src="generated"/>
    <c:component name="Meeting Scheduler"
                        src="generated: renamed"/>
    < goal name="Select Participants"> ... </goal>
    < goal name="Collect Timetables"> ... </goal>
    < goal name="Choose Schedule"> ... </goal>
  </goal>
</view>
```

**B** Generated features

```
<view xmlns:g="urn:goals" xmlns="urn:features"...>
  < g:goal name="Schedule Meeting">
    <g:refinement type="goal" value="AND"/>...
    <e:annotation type="feature" value="SYS"/>...
    < feature name="meeting scheduler"
           src="renamed"> <refinement type="feature"
           value="AND" src="generated"/>
      <cardinality value="1" src="generated"/>
      < g:goal name="Select Participants">
        < feature name="participants selector"
              src="renamed"> ...</feature>...</g:goal>
        < g:goal name="Collect Timetables">s
          < feature name="timetable collector"
              src="renamed">...</feature>...</g:goal>
        < g:goal name="Choose Schedule">
          < feature name="timetable collector"
              src="renamed">...</feature>...</g:goal>
        < feature name="message decrypter"
              src="refactored"
              origin="Decrypt Received Message">
...<feature/> </feature> ...</g:goal> </view>
```

Figure 5-12. Traceability support in representations of annotated goal models.

We use the generated feature model view to illustrate the support for traceability (Figure 5-12B) among models. The default namespace here is `features` and the nesting structure for feature models is mostly generated from the nesting structure of the corresponding goal model. However, some features (e.g. "message decrypter") can correspond to several original goals to support the refactoring of the connector-component model. In this case, the original goal with which it was associated is indicated. During design activities, features not corresponding to any goal can be added/removed. However, such design changes cannot remove existing traceability links including the ones implied by the sibling XML elements. Deleted design elements are still kept in the model with a "deleted" `src` attribute. As the `src` attribute maintains the traceability between a goal and a design element in the generated model, the link between two design views can be maintained. For example, in the above document, one can obtain the traceability link between a "Meeting Scheduler" component and a "meeting scheduler" feature as they are both mapped from the Schedule Meeting goal.

The XMI representation of the above metamodel represents the elements of a unified model in a single namespace. For each design view, the generation procedure is implemented as an XSLT [XSLT07] stylesheet that transforms the unified model into an XML design document that can be imported by the corresponding visual tool. A generated document separates the concerns of various views into different namespaces. The default namespace deals with the primary design view, whereas the relationships between adjacent tags of different namespaces capture the traceability among the corresponding design views.

A case study was conducted [YLLM08a], [YMLL05] to test the generation and maintenance of the feature models, statecharts and connector-component diagrams. It used an open source implementation of an email system that was previously used in an approach to reverse-engineer requirements from implemented systems [YWML05]. However, the purpose of this study differs from that of [YWML05], which was to recover as-is requirements from the code. Rather, we compared the implementation with respect to the goal model refining the goal Prepare an Electronic Message that appears in [HLM03] in order to check (1) whether we could reuse any components in the email system to implement the design views that were generated from goal models; (2) whether the email system could cover all the variability needed by our goal model; (3) whether our final design could be traced back to the original requirements models.

## 5.5 Conclusion and Related Work

There is growing interest on the topic of mapping goal-oriented requirements to software architectures. Brandozzi et al. [BP01] recognized that requirements and design were respectively in problem and solution domains. A mapping between goals and components was proposed for increasing reusability. More recent work by van Lamsweerde [Lam03a] derives software architectures from the formal goal model specifications using various heuristics. Specifically, the heuristics discover design elements such as classes, states and agents directly from the temporal logic formulae that define the goals. Complementary to their formal work, we apply light-weight annotations to the goal model in order to derive design views: if one has the formal specifications for each goal, some heuristics provided in [Lam03a] can be used to find the annotations we need, such as system/non-system goals, inputs/outputs and dependencies among the subgoals.

Variability within a product-line is another topic receiving considerable attention [CN01], [CE00]. However, previous research has not addressed the problem of linking product family variability to stakeholder goals and intentional variability. Closer to our work, [HP03] proposes an extension to the use case notation to allow for variability in the use of the system-to-be. More recently the same group tackled the problem of capturing and characterizing variability across product families that share common requirements [BLP05].

We maintain traceability between requirements and the generated design using concepts from literate programming and model-driven development. In *literate programming* [Knu84], any form of document and program can be put together in a unified form, which can then be tangled into an executable program with comments or into a document with illustrative code segments. In this regard, our annotated goal model is a unified model that can be used to derive the preliminary views with traceability to other views. In *model-driven development* (MDD), code can be generated from design models and a change from the model can propagate into the generated code [Sel03]. To prevent a manual change to the code from being overridden by the code generation, a special *not generated* annotation can be manually added to the comments of the initially generated code. We use MDD in a broader sense to allow designers to change the generated preliminary design views and to manually change the *not generated* attribute of a generated design element. Comparing to deriving traceability links through information retrieval [HDS06], our work proposes a generative process where such links are produced by the process itself and recorded accordingly.

In [CL04], softgoal models that represent non-functional requirements were associated with the UML design views. Here, we focus on establishing traceability between functional requirements and preliminary designs.

Most GORE approaches bind the intentional variability elicited at the requirements level before proceeding to software design. We, on the other hand, believe that it is beneficial to preserve this variability downstream in the software development process. While aiming at selecting the best way to achieve system goals may produce a requirements specification that is truly optimal, changing circumstances may lead the system built based on that specification to not fulfill its requirements or to become suboptimal. Similarly, often there is no single system configuration that will suit every user. Finally, systems may fail due to the lack of alternative means of achieving certain failed goals in their implementation: they will not be able to easily reconfigure themselves (or be reconfigured) to use a different alternative to avoid the problem. The approach presented in this chapter helps alleviate the above difficulties by proposing to preserve intentional variability in the solution domains, i.e., in the artefacts produced in the subsequent software engineering activities. This idea provides the foundation for our requirements-driven approach for designing customizable/configurable and adaptable/adaptive software systems. We use a small set of mappings linking goal model patterns to the appropriate patterns in the target notations and exploit the hierarchical nature of goal models to combine the model fragments. What results is a set of variability-preserving transformations that given HV goal models representing variability in the problem domain (together with the appropriate annotations capturing additional domain details) produce target models that represent this variability in the solution domain. In addition to preserving requirements-level variability, the choices among the alternatives in the generated models (e.g., in terms of the systems' behaviour, configuration, etc.) can be linked back to the selections within the appropriate goal model variation points. We illustrate three such transformations in this chapter. We generate feature models, high-variability behavioural specifications using the statecharts notation, and flexible architectural models. These representations are connected to variation points of the source goal models through parameterization. Thus, intentional variability present in the source goal models is preserved as *configuration variabilit*y in feature models, *behavioural variability* in statecharts, and *structural variability* in connector-component models. This way the selection of a particular configuration in the goal model will result in the appropriate behaviour or architecture being automatically selected in the generated models. This can be done during the instantiation of the system variant or potentially at runtime. Note that in general the generated models are preliminary in the sense that not all of the details of the target notations can be captured at the requirements level using goal models. These preliminary models will need to be augmented with additional details. Tool support discussed in this chapter can help in maintaining traceability between the solution-oriented models and the original goal models while allowing the former to be augmented with additional details. On the other hand, in this chapter, we show that some advanced feature model constraints can be easily generated from goal models.

The ideas discussed in this chapter are the basis of the research presented in the subsequent chapters. For instance, in Chapter 6 we use the design of customizable/configurable business processes as a case study for evaluating the approach presented in this chapter. There, we generate executable high-variability BP specifications that can be configured using the original goal models. Also, in Chapter 7 we argue that high-variability designs can be used as the basis for developing adaptive/autonomic systems. Furthermore, Chapter 8 discusses awareness requirements as a source of requirements for adaptive systems as well as the starting point for the design of the feedback loop-based adaptive infrastructure. There, we also assume to have high-variability software implementations systematically derived from high-variability requirements models. Overall, the approach discussed here allows for the use of goal models for analysis and the selection of software behaviour alternatives at the time of system instantiation or even at runtime with the help of goal analysis techniques [SGM04], [GMNS02].

# Chapter 6 Requirements-Driven Design and Configuration Management of Business Processes

**Acknowledgement**: This chapter is based on [LYM07]. We acknowledge the contribution of Yijun Yu to the implementation of the prototype infrastructure discussed in this chapter.

## 6.1 Introduction

In the previous chapter we presented a number of variability-preserving transformations to support the systematic generation of flexible design-level models (as well as other solution-oriented representations such as feature models). The benefits of the approach include the ability to use high-level goal models to analyze the available alternatives and select the appropriate ones for system configuration/reconfiguration/customization at various times, including at instantiation time and at runtime. To further test the ideas described Chapter 5, here we look at the domain of business process modeling and management. BPs are a good fit for our requirements-driven approach. Business processes are conceptual representations of business activities in organizations and as such can be modeled at high level. Systematic requirements-driven development of business processes is very beneficial to companies since in most cases their success depends on how their BPs fit the business practices, how reliable and flexible they are, etc. Using BPs we can represent systems at a level that is accessible to both the IT staff responsible for the implementation and maintenance of these systems as well as to business users. In this chapter, we use the approach from Chapter 5 to create a high-variability *executable* BP specification from HV goal models in a semi-automatic way.

At present, process orientation is a dominant paradigm for businesses [DAH05]. There are many definitions of what a business process is, but in general a BP is seen as a collection of activities that achieves some business *purpose* or *objective* aiming to create value for customers. So, business processes specify ways to achieve business *goals*. Thus, it seems to be natural for business process modeling methods to include facilities for modeling these goals. However, relatively few approaches explicitly capture, refine and analyze business goals (e.g., [KL99],

[KK97]). Most leading BP modeling approaches capture processes at a workflow level, in terms of activities, flows, etc. (e.g., [BPMN04]).

Due to the need to accommodate changing business priorities as well as business cases with varying characteristics (e.g., customers with different preferences), business process specifications need to be flexible as well as capable of being configured and reconfigured appropriately. Currently, techniques as diverse as business rules and late modeling are used for changing BPs. However, these approaches are usually quite low-level and the possible configurations are not explicitly evaluated with respect to business goals and priorities. Thus, it is hard to select process alternatives with desired non-functional characteristics. Additionally, most of these methods require extensive knowledge of the process and, possibly, the modeling notation to be effectively applied, thus making it difficult for non-technical users to configure BPs.

To alleviate the above difficulties, we are proposing a systematic business requirements-driven method for configuration of high-variability business processes at a high level, in terms of business priorities. In our approach, we start by employing goal models to capture and refine business goals as well as to explore and analyze the variability (the various ways these goals can be attained) in the business domain. Quality attributes such as customer satisfaction serve as the selection criteria for choosing among BP alternatives induced by the goal models. These high-variability goal models are then used in a semi-automatic variability-preserving transformation to generate customizable executable business processes (in our case study we use the Business Process Execution Language (BPEL) [BPEL07]). Through the preserved traceability links to goal models, the executable processes can be configured based on qualitative preferences of stakeholders. Automated analysis of the models is used at design time or at runtime to identify process alternatives that best match these preferences. A GUI tool for capturing user preferences and a prototype runtime infrastructure are also provided.

## 6.2 Goal Model-Based Customization and Configuration

There has been interest in applying goal models in practice to configure and customize complex software systems. In [HLM03], goal models were used in the context of “personal software” (e.g., an email system) specifically to capture alternative ways of achieving user goals as a basis for creating highly customizable systems that can be fine-tuned for each particular user. The

Goals-Skills-Preferences approach for ranking alternatives is also proposed in [HLM03]. The approach takes into consideration the user's preferences (the desired quality attributes) as well as the user's physical and mental *skills* to find the best option for achieving the user's goals. This is done by comparing the skills profile of the user to the skills requirements of various system configuration choices. For example, for the user who has difficulty using a computer keyboard, the configurator system will reject the alternatives that require typing in favour of voice input.

Goal models can also be used for configuring complex software systems based on high-level user goals and quality concerns. Liaskos et al. [LLWY05] propose a systematic way of eliciting goal models that appropriately explain the intentions behind existing systems. In [YLLM05], Yu et al. show how goal models can be used to automatically configure relevant aspects of a complex system without accessing its source code.

## 6.3 Requirements-Driven Business Process Configuration

In this section, we describe our technique for business process modeling and configuration. It is requirements-driven and is motivated by the lack of support in most current BP modeling approaches for high-level, intentional configuration of business processes. The approach involves the modeling and analysis (using quality criteria) of alternative ways of achieving business goals with subsequent generation of executable business processes that preserve the variability captured at a goal level. The assumption behind this approach is that in the business domain where it is applied, the characteristics of business cases demand tailored business process variants. Below, we briefly outline the steps of the process and highlight the responsibilities of various actors while the subsequent sections describe the process in detail:

Table 6-1. Overview of the process steps.

| | **Responsible Role** | **Description** | **Artefact Produced** |
|---|---|---|---|
| 1 | Business Analyst (BA), Business Users | Capture and refine the goals of the business process with emphasis on variability. | High-Variability (HV) Goal Model |
| 2 | BA, Requirements Engineer | Enrich the model with control flow and I/O annotations. | Annotated HV Goal Model |
| 3 | BA | Analyze BP alternatives, remove | Annotated HV Goal |

| | | | |
|---|---|---|---|
| | | infeasible ones. | Model |
| 4 | Automated | Generate initial High-Variability BPEL specification from HV Goal Model. | Initial HV BPEL process |
| 5 | BPEL / Integration Developer | Complete the HV BPEL process, select partner Web Services, deploy process. | Executable HV BPEL process |
| 6 | Business Users | Select prioritizations among available quality criteria. | BP Preferences, Configured Goal Model |
| 7 | Automated | Select the best BP configuration matching user preferences. | BP Configuration |
| 8 | Automated | Create BP instance with the selected configuration, execute it. | Configured BPEL process |

### 6.3.1 Business Process Design with Goal Models

Using goals for business process modeling is not a new idea. A number of different goal modeling notations have been used for this [KL99], [KK97]. In addition, goal models have shown to be a convenient notation for the elicitation, modeling, and analysis of variability in the context of software development, configuration, and customization [LLWY05], [YLLM05]. In our approach, we use high-variability goal models to capture *why* a business process is needed – its purpose or goal – and the many different ways *how* this goal can be attained. Business process alternatives implied by the models are then evaluated with respect to their quality (non-functional) attributes.

First, let us look at a distribution company selling goods to customers. We use this example throughout the remainder of the chapter to demonstrate our approach. The company gets its products from wholesalers and sells the goods to customers (see Figure 6-1). It does not have any retail stores, so it receives orders though phone, fax, and, possibly, a web site and ships products using a shipping company. To model a BP in our approach we first identify its business goal (Supply Customer). This goal becomes the root of the goal model. It is then refined using AND/OR decompositions until the resultant subgoals can be delegated to either human actors or software services. In our example, Supply Customer is AND-decomposed into a number of goals including Get Order, Process Order, and Ship and Bill [order]. Some of the subgoals have alternative solutions that are captured, as usual, using OR decompositions. For example, to ship an order, one can achieve either the Ship Express goal or the Ship Standard goal.

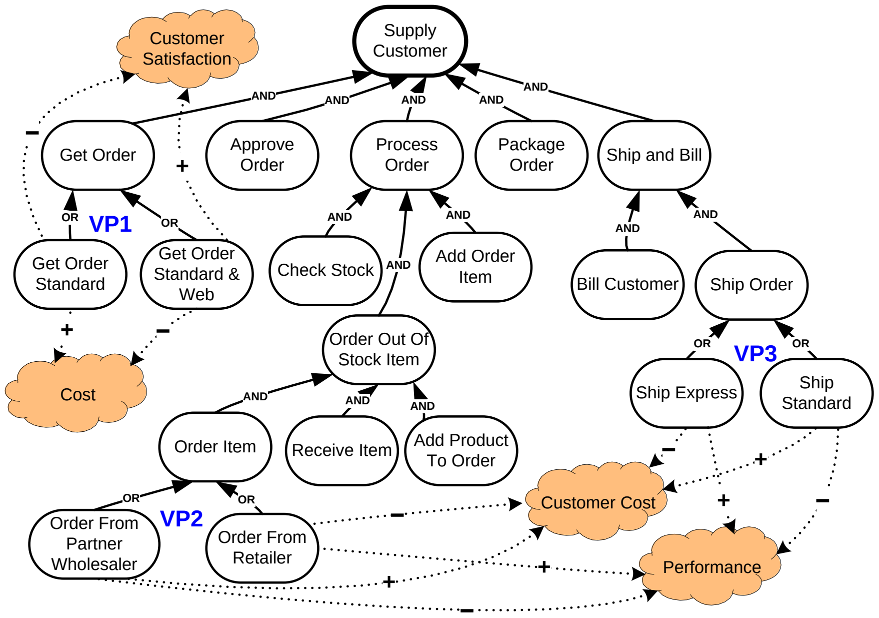


Figure 6-1. The refinement of the business goal Supply Customer.

In our example, the four top-level desired qualities (represented by softgoals) are Customer Satisfaction, [Minimize distributor] Cost, [Minimize] Customer Cost, and Performance. Clearly, express shipping is fast, but expensive, thus it helps the softgoal Performance while hurting Customer Cost. Similarly, providing a web site for order submission (Get Order Standard & Web) may be more expensive for the distributor (thus the negative link to Cost), but contributes positively to Customer Satisfaction. In all, the goal model in Figure 6-1 shows eight alternative ways for fulfilling the goal Supply Customer. As mentioned in the previous chapters, it is easy to verify that generally the number of alternatives represented by a typical goal model depends exponentially on the number of OR decompositions (labelled as variation points “VP1” through “VP3” in Figure 6-1) present in the goal model (assuming a “normalized” goal model where AND and OR decompositions are interleaved). As such, goal models make it possible to capture during requirements analysis – in stakeholder-oriented terms – all the different ways of fulfilling top-level goals. In terms of the elicitation of this variability, in the context of business process design, we can employ a method like the systematic approach for elicitation and modeling the variability in the problem domain with the help of high-variability goal models that is presented in [LLYY06]. Now, if one were designing a flexible, customizable implementation for a process, it would make sense to ensure that the implementation is designed to accommodate most or all ways of fulfilling top-level goals (i.e., delivering the desired functionality), rather than just some.

The ability to use goal reasoning algorithms to rank the alternative ways of attaining business goals that are represented in goal models by comparing their overall contributions to softgoals is another advantage of using goal models for BP design.

Let us have a detailed look at the Supply Customer process. First, customer orders are received either through phone, fax, or the web. After approving an order, the distributor processes the order by checking if it has all the ordered goods in stock. If so, each product is added to the order. If some item is not in stock, it is ordered from either a wholesaler or a retailer. Ordering out of stock goods through the usual channel from a wholesaler is cheaper (positive contribution to Customer Cost), but requires more time (negative contribution to Performance), while ordering the goods from a nearby retailer to complete the order has the opposite contributions to these softgoals. After an order is packaged, it is shipped (using either the express or the standard shipping method) while the customer is sent a bill.

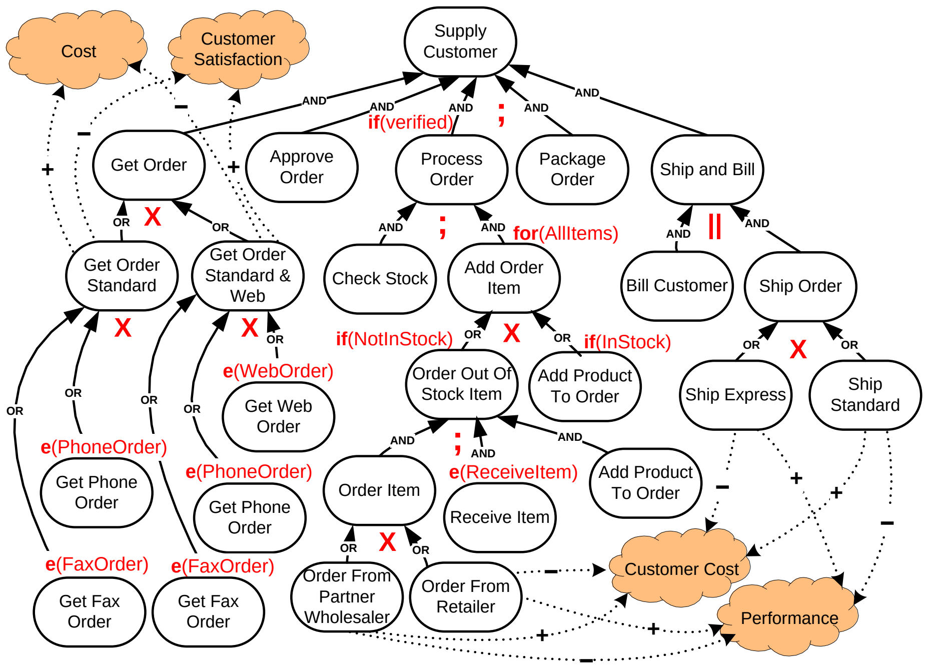


Figure 6-2. A goal model for the Supply Customer business process.

We are placing a special emphasis on business process variability since explicitly representing the space of alternatives using goal models allows for a systematic analysis and comparison of the various ways of achieving high-level business goals (i.e., the various BP alternatives). Whenever there is a number of different ways to achieve some business goal, the modeler uses

OR decompositions to capture that fact. Some of these alternatives contribute differently to the non-functional business concerns such as Customer Satisfaction, [Minimize] Cost, etc. represented as softgoals in the model. We describe how these quality criteria are used in selecting the appropriate process configurations in Section 6.3.3.

### 6.3.2 Enriching Goal Models for BP Modeling

As discussed in Chapter 5, when we use variability-preserving transformations to produce solution-oriented HV models from requirements models, we often use annotations to capture the details of the problem domain that are necessary to produce meaningful target models. Since we are interested in the automated execution of business processes, the target notation that we use in this chapter is quite low-level. Therefore, we would like to be able to capture more information about BPs than the basic goal models allow. A few annotations are used for this purpose. Note that the annotations presented here are not required to be formal. We use the following control flow annotations when employing goal models to represent business processes:

- Parallel ("||") and sequential ("**;**") annotations can be used with AND-decomposed goals to specify whether or not their subgoals are to be achieved in a temporal order.
- By default, in goal models, OR decompositions are inclusive. Exclusive OR decompositions are marked with the "X" annotation. All of the OR decompositions in our example in Figure 6-2 are exclusive.
- Conditions ("`if(condition)`") indicate the necessary conditions for achieving subgoals. For example, in Figure 6-2, the goal Order Out Of Stock Product is achieved only if the item is not already in stock.
- Loops ("`while(condition)`" or "`for(setOfItems)`"). For instance, the goal Add Order Item must be achieved for all items in the order.
- Event handlers or interrupts ("`e(Event)`"). In Figure 6-2, the arrival of customer orders through fax, phone, or web is modeled by the events (e.g., `e(PhoneOrder)`) that trigger the achievement of the appropriate goals.

In addition to the above annotations, modeling of input/output parameters of goals is also important for BP modeling. Identifying inputs and outputs during the analysis of a business domain helps in determining resource requirements for achieving goals as well as for the sequencing of the goals. The types of inputs and outputs can also be specified. While optional, the input/output types can be used to generate detailed specifications for messages and service interfaces in a BP implementation. For example, Figure 6-3 shows a parameterized fragment of the Supply Customer goal model. The parameters are specified inside goal nodes and the output parameters are identified with the star ("*") symbol. Deliberating about which resources/data are required for the attainment of a goal and which are produced when the goal is achieved can frequently help in identifying important process details that are easy to miss otherwise. For instance, Figure 6-3 adds the subgoal Pick Order Bin, which picks a location where ordered items are physically stored before being packaged.

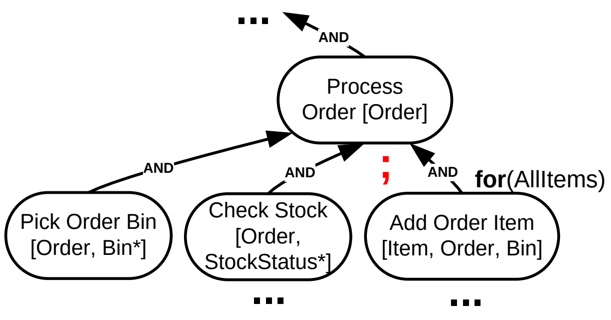

...
AND
Process Order [Order]
AND
AND
;
AND
for(AllItems)
Pick Order Bin [Order, Bin*]
Check Stock [Order, StockStatus*]
Add Order Item [Item, Order, Bin]
...
...

Figure 6-3. Adding goal parameters.

## 6.3.3 Specifying Goal Model Configurations

In goal models, there exist OR decompositions where the selection of alternatives is driven by data or events. For example, in Figure 6-2 the OR decomposition of the goal Get Order Standard is event-driven as the choice depends on the way the customer submits an order. Similarly, the choice for achieving the Add Order Item goal depends on whether the item is in stock. However, there are other OR decompositions with alternatives, whose selection is not dependent on data/events. These are intentional OR decompositions or variation points (the data-/event-driven OR decomposition are not considered VPs as they cannot be used to configure processes). For the purpose of business process or system configuration, we can consider VPs as *preference-driven* OR decompositions. In the example in Figure 6-2, these variation points are: Get Order, Order Item, and Ship Order. From the point of view of the functionality of a business process, the achievement of any of the alternative subgoals of these VPs is exactly the same. The difference is in the way these choices contribute to the quality attributes of the process. These VPs play a central role in our business process configuration approach as they allow the selection of the best way to meet quality constraints of the stakeholders while delivering the required functionality of business processes. Thus, softgoals act as (possibly *conflicting*, as seen in our example) selection criteria for choosing the right BP alternative based on the priorities (among softgoals) of process owners, customers, etc.

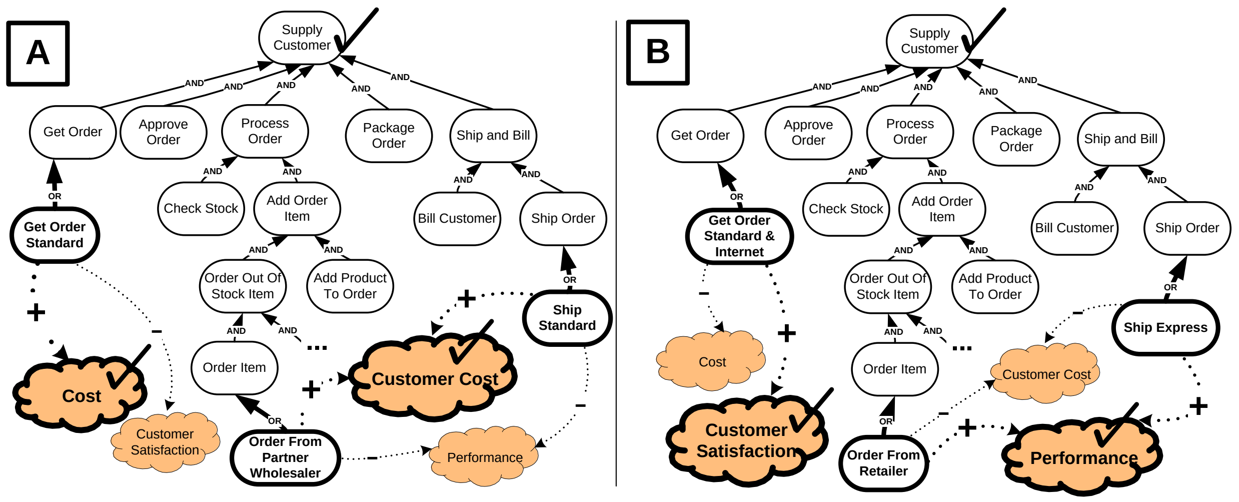


Figure 6-4. Two alternative goal model configurations.

To illustrate the above discussion, Figure 6-4 shows two alternative configurations of the process Supply Customer. These configurations are the result of applying the top-down goal reasoning algorithm of [SGM04] to the model in Figure 6-2. The checkmarks indicate the highlighted (soft)goals, whose achievement we are interested in – the input to the algorithm (another input is the relative ranking of the softgoals, which we assume to be the same here). The first configuration (Figure 6-4A) is where the Cost of running the process for the distributor and Customer Cost are the top priorities. The highlighted VP decisions contribute positively to the selected softgoals. This configuration includes, for instance, Ship Standard as it is cheaper. If, however, Customer Satisfaction and process Performance are the top priorities, then the configuration changes to the one in Figure 6-4B. Thus, high-variability goal models provide a high-level view of processes with the ability to (automatically) generate BP configurations based on preferences of stakeholders expressed as prioritizations among quality criteria. These features greatly simplify the task of configuring business processes by non-technical users as these individuals can configure processes in terms of user-oriented abstract qualitative notions such as customer satisfaction, etc.

It is easy to notice that in our example, a goal model can be configured by multiple stakeholders, both from the point of view of the process owner (the distributor) by prioritizing among the Cost and the Customer Satisfaction softgoals and from the point of view of the customer by prioritizing among Customer Cost and Performance. This allows the stakeholder that owns the process to partially configure it based on that stakeholder's own preferences (i.e., the *binding* some of the variability) while leaving other VPs unbound for the customers, partners, etc.

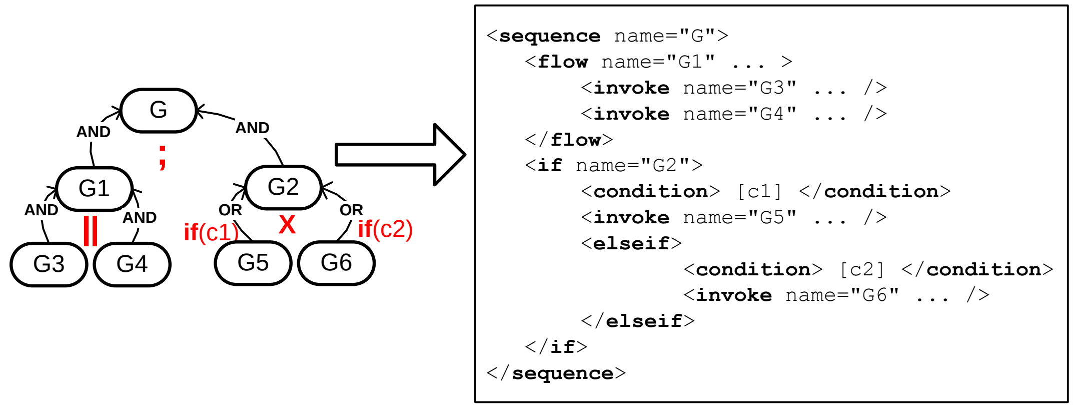
Note that the alternatives deemed not acceptable by the process owner (e.g., due to being too costly) can be removed from goal models, thus reducing the BP variability before the generation of executable BP models.

### 6.3.4 Generating Flexible Executable Business Processes

As we have just shown, goal models can be a useful tool for high-level configuration of business processes based on stakeholder prioritization among quality criteria. The above techniques can be used to develop, analyze, and configure BP models at design time. However, we would also like to be able to use the high-variability goal models as a starting point for the development of executable business processes that preserve the variability found in the source goal models as well as for configuring these BPs though the appropriate traceability links.

To this end, we use the general idea for variability-preserving transformation of requirements models into solution-oriented models that we discussed in Chapter 5. Here, we present a method for using goal models to assist with the development and configuration of high-variability (flexible) BPEL processes. Unlike some workflow-level notations such as BPMN [BPMN04], our goal modeling notation is highly structured, with goals organized in refinement hierarchies. This makes it possible to generate BPEL processes (albeit lacking some low-level details) that are easily readable by humans and are structured after the respective goal models. The BPEL code generation is *semi-automatic* and the generated code, while not immediately executable and thus needing to be completed (mainly due to the fact that we do not require conditions in annotations to be formalized), nevertheless provides valuable help in producing an executable

BP based on the source goal model. The code is to be further developed by integration developers, who will also be selecting/designing Web services to be used by the process.



```
<sequence name="G">
   <flow name="G1" ... >
        <invoke name="G3" ... />
        <invoke name="G4" ... />
   </flow>
   <if name="G2">
        <condition> [c1] </condition>
        <invoke name="G5" ... />
        <elseif>
                <condition> [c2] </condition>
                <invoke name="G6" ... />
        </elseif>
   </if>
</sequence>
```

Figure 6-5. Example of WS-BPEL 2.0 code generation.

Since BPEL is a workflow-level language, only activities that are executed by human actors or software systems (Web services) are represented in BPEL specifications. On the other hand, using goal models, we start modeling from abstract high-level goals and refine them into goals that can be assigned to humans or software. Thus, leaf-level goals correspond to the actual work that is done within a BP, while higher-level ones provide the rationale for why this work has to be done and how it relates to the ultimate purpose of a process. Thus, non-leaf goals do not create basic BPEL activities, but since they are used to group lower-level goals based on their decomposition types (AND/OR) and control flow annotations, they help in generating the corresponding BPEL control flow constructs. We start BPEL generation from the root goal and recursively traverse the goal tree until we reach leaf goals.

We now present some of the goal model to BPEL 1.1 or 2.0[3] mappings through the example in Figure 6-5, which shows a generic annotated goal model fragment. The root goal `G` has a sequential AND refinement, so it corresponds to the `sequence` operator in BPEL. `G1` has a parallel AND refinement, so it maps to the `flow` construct. `G2` has a data-driven XOR refinement (note the annotations), so it generates the `if-elseif` (BPEL 2.0) or the `switch` (BPEL 1.1) operator. Note that the conditions `c1` and `c2`, which are informal descriptions in the

[3] While in our case study we generated BPEL 1.1 processes, WS-BPEL 2.0 [BPEL07] allows for simpler, more natural mapping from annotated goal models.

goal model, will be replaced with the appropriate conditions by a BPEL developer. The leaf goals correspond to Web service invocations. This is how enriched goal models are used to generate the overall structure of a BPEL process.

While we abstract from some of the low-level BPEL details such as correlations, with the information captured in the annotated goal models, we also generate the following aspects of BPEL/WSDL specifications (here we show the main mappings while the mapping of VPs is discussed later in Section 6.3.5):

- We do an initial setup by defining the appropriate interface (`portType`), etc. for the process. A special `portType` for invoking the process and providing it with the (initial) configuration is also defined.

- An event-driven OR decomposition (e.g., Get Order Standard in Figure 6-2) maps into the `pick` activity with each alternative subgoal corresponding to an `onMessage` event. Since each such event must match an operation exposed by the process, an operation with the name of each subgoal is added to the `portType` of the process. A message type for the received event is also added to the process interface. A BPEL developer must define the message as the event annotations specified at the requirements level usually lack the required message details. The activities that are executed for each `onMessage` event are the BPEL mappings of the subtrees rooted at the subgoals in the decomposition.

- A conditional/loop annotation for a goal G is mapped to the appropriate BPEL construct (e.g., `if-elseif` or `switch`, `while`, etc.) with the activity to be executed being the result of mapping the goal model subtree rooted at G into BPEL. The formal conditions currently have to be specified manually.

- Leaf-level goals map into Web service invocations. The information in the goal model helps in defining the interface for the Web services invoked by the BP. We define appropriate WSDL messages based on input/output parameters of these goals. If data types are omitted from the goal model, they have to be supplied by a developer.

- Softgoals are used as the evaluation criteria in the configuration process and thus do not map into the resulting BPEL specification.

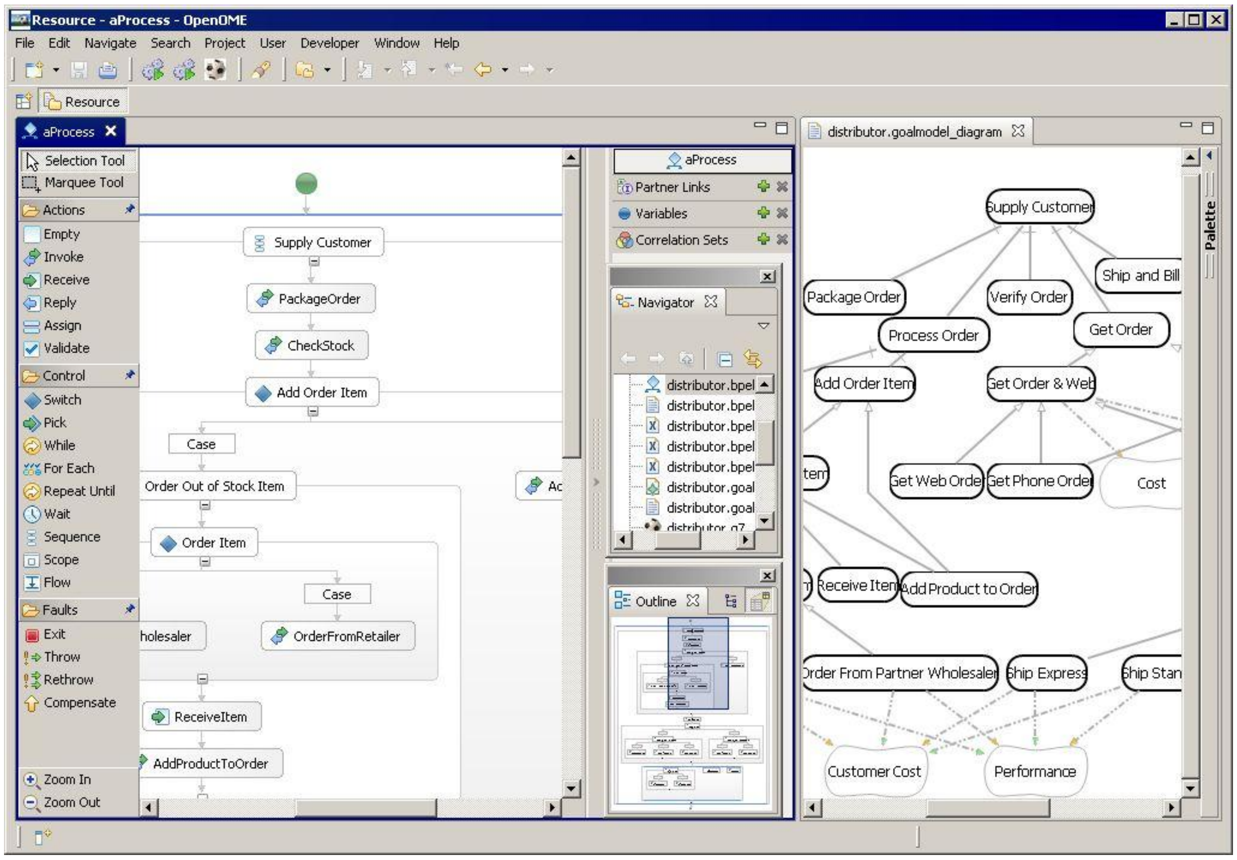


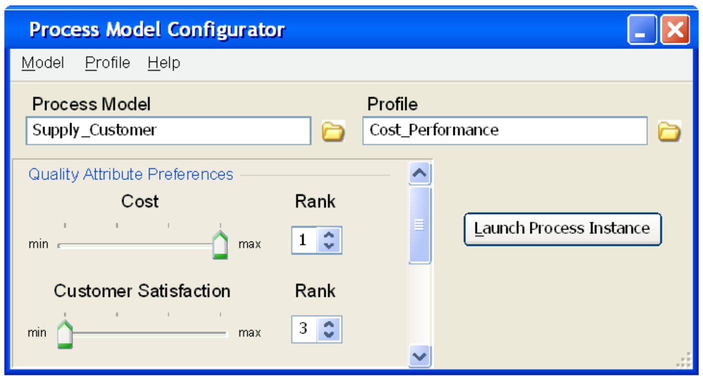
Figure 6-6. OpenOME being used to design a business process.

The main idea behind the generation of high-variability BPEL processes is the preservation of BP variability captured in goal models. As we have shown above, data- and event-driven variability is directly preserved through the appropriate mapping to BPEL. Additionally, we need to preserve the preference-driven VPs in the executable BPs since they are the main vehicle for process configuration based on stakeholder preferences. In our approach, for each preference-driven VP we generate a BPEL `switch` construct (or `if-elseif` if using BPEL 2.0) where each case (branch) corresponds to an alternative subgoal (e.g., Ship Order in Figure 6-2 will produce the cases for Ship Express and Ship Standard). The condition in each case checks to see if the case is the current choice for the VP by comparing the name of the alternative subgoal it corresponds to (e.g., “Ship Express”) to the string extracted from the current BP configuration (see the next section for details), thus ensuring the correct branch is taken. The activities executed in each case are automatically generated and represent the BPEL mapping of the alternative subgoals of the VP. A VP also gets a name from the corresponding goal node (we assume that VP names are unique).

Figure 6-6 shows our Eclipse-based goal modeling and analysis tool OpenOME [OOME07] being used to design business processes with both the goal model (right pane) and the BPEL (left pane) visualizations.

## 6.3.5 Quality-Based Business Process Configuration

Once a High-Variability BPEL process is fully developed and deployed, its instances can be configured by users through the prioritization among its associated quality criteria. This task has two elements. First, we elicit user preferences and generate the corresponding process configuration. Second, an instance of a BP has to be provided with this configuration. Let us look at these two activities in more detail.



Process Model: Supply_Customer
Profile: Cost_Performance
Quality Attribute Preferences
Cost — Rank: 1
Customer Satisfaction — Rank: 3

Figure 6-7. BP preference configuration tool.

There are several ways to specify user preferences in our approach. First, users can utilize OpenOME to specify which softgoals they want satisficed in a process and run a top-down analysis algorithm (similar to what we did in Figure 6-4). The result will be a particular BP configuration that best suits the user. Another possibility is to use the GUI tool (see Figure 6-7) that simplifies the task even more by only exposing quality attributes of a process and by allowing users to specify the partial ordering of the attributes in terms of their importance (Rank) as well as their expected satisficing level (with convenient sliders). Multiple profiles can be created for a particular BP model – for varying market conditions, customers, etc. Behind the scenes, a preference profile is converted into a goal model configuration (using the same goal reasoning algorithm of [SGM04]). The tool can then create instances of processes with the desired configuration.

Another part of our prototype BP configuration toolset is the Configurator Web service. This service implements a subset of the functionality of OpenOME, mainly the top-down reasoning engine and a persistent knowledge base for storing process configurations. It is designed to communicate these configurations to appropriate BP instances at runtime. The main operations of the service are as follows:

- *`registerProcess`* is used to associate a unique `processID` parameter with the endpoint of a deployed high-variability process (both are inputs).
- *`launchProcessInstance`* is be used by the GUI tool to create and run an instance of a process. Inputs are `processID` and a goal model configuration. A new instance of a BP identified by `processID` is created. It is given an `instanceID`, which uniquely identifies the process instance together with its configuration so that it is possible to evolve BP configurations independently. The configuration is stored in a knowledge base. The configuration and the `instanceID` are communicated to the BP instance.
- *`getConfiguration`*, which can be used by process instances to get their configurations from the Configurator Web service. The input is an `instanceID` and the output is the current configuration for that process instance. This operation can be used to get an updated configuration for a process instance at runtime.

The configuration provided to process instances is a list of variation points and the name of the selected subgoal in each of them. Below is an example:

```
<FullConfig>
  <VPConfig>
        <VP> ShipOrder </VP>
        <Selection> ShipStandard </Selection>
  </VPConfig>

  …
</FullConfig>
```

Then, XPath [XPAT07] queries are used to extract the configuration. For example, the query `/FullConfig/VPConfig[VP="ShipOrder"]/Selection` extracts the configuration for the variation point `ShipOrder`. The result is matched with the appropriate case in the switch construct corresponding to the variation point as described in the previous section. Thus, the executing process becomes configured according to the user preferences specified in terms of priorities among quality criteria associated with the business process.

## 6.4 Discussion

Most popular BP modeling approaches such as BPMN [BPMN04] or EPCs [KNS92] are workflow-level notations. They do not allow the analysis of process alternatives in terms of high-level quality attributes or business goals and thus do not provide traceability of BP alternatives to requirements. There are, however, BP modeling approaches that explicitly capture and refine business goals (e.g., [KL99], [KK97]). Unlike our approach, these notations do not model process variability or the effect of alternatives on quality attributes. While some research has focused on variability in business process models [SP06], our approach centers on capturing and analyzing variability at the requirements level. Similarly, research on configurable BPEL processes (e.g., [KLB05]) so far mostly concentrated on low-level configurability that may not be visible to process users.

A number of approaches based on the Tropos framework [BPGM04] applied ideas from requirements engineering and agent-oriented software engineering to BPEL process design [KPR04] and SOA architecture design [LM04]. Both these approaches give heuristics on creating BPEL processes based on requirements models, but fall short from providing semi-automatic generation procedures. Likewise, BP variability is not explored. Nevertheless, we believe that agent-oriented modeling techniques of Tropos are useful for BP modeling and analysis and are currently working on integrating them into our approach. Similarly, we are looking at supporting context-based softgoal prioritization where the preferences change depending on the characteristics of business cases. For instance, in our Supply Customer example, the process owner may want to set Customer Satisfaction to be the top priority for high-valued customers.

The drawbacks of the approach presented here include the need to explicitly model and analyze process alternatives as well as the fact that the qualitative analysis of alternatives may be too imprecise and subjective.

Looking at the way we do process configuration in the approach presented in here, we can see that our flexible BPEL process specification has an implicit population of invariants (as discussed in Section 2.3.3) since the knowledge of the variants/alternatives is not present in the system and new alternatives cannot be easily added. Also, it has an external binding of variability since the system itself (i.e., the high-variability BPEL process) cannot decide on its own which alternative to use. Based on this, one can easily conclude that this system is inflexible and cannot possibly exhibit self-adaptive behaviour. However, the high-variability BPEL process is not a complete system. It is just the *managed element* (in the language of autonomic computing [KC03]) with the adaptive functionality provided by an external autonomic manager that has the knowledge of the available alternative and can determine the best configuration based on high-level user input. Thus, we need to consider the complete autonomic element consisting of both the manager/controller and the managed element (the BPEL process and the configurator Web Service) to evaluate its capabilities for runtime configuration and adaptation.

## 6.5 Conclusion

We have presented here an approach for requirements-driven design and configuration of business processes. Goal models are used to capture and refine business goals with the emphasis on identifying alternative ways of attaining them while (possibly conflicting) quality constraints are used to analyze and select appropriate process alternatives. Then, given an annotated high-variability goal model, a variability-preserving procedure generates a well-structured high-variability WS-BPEL specification (with programmers needing to fill in details of data handling, to define conditions and some other aspects of the process), which can be configured given high-level user preferences. A prototype system for preference profile specification and BP configuration is discussed.

The benefits of the approach include the fact that BPs can be automatically configured in terms of criteria accessible to non-technical users, thus greatly simplifying process configuration. The method helps in transitioning from business requirements analysis to BP design and implementation by allowing to gradually increase the level of detail in process models and by providing a semi-automated variability- and structure-preserving procedure for generation of executable business processes. The approach is also helping to maintain the processes' traceability to requirements.

This chapter is presented as a case study aimed at verifying that the ideas discussed in Chapter 5 of this thesis can be indeed implemented to generate a high-variability instance of a model in a well-known notation from a high-variability goal model using a systematic and semi-automatic goal-driven process. Moreover, here we create not just a high-variability design view, but an executable representation of a business process. We also show the value of the traceability of flexible BPE specifications to goal models through parameterization as it supports a novel approach for business process configuration using high-level user-oriented terms. This confirms the usefulness and implementability of the idea that HV goal models can be used to analyze and manage flexible software that has been systematically designed using the approach presented in Chapter 5 of this thesis.

# Chapter 7
# Requirements-Driven Design of Adaptive Application Software

**Acknowledgement**: This chapter is based on [LLMY05] and [LYLM06]. We acknowledge the contributions of Sotirios Liaskos and Yijun Yu to the research presented here.

## 7.1 Introduction

In this chapter, we show that high-variability software designs developed from high-variability goal models can become the foundation for the development of adaptive or autonomic systems. Note that this research has been performed in the context of autonomic computing, which is an approach to adaptive systems design that employs the architecture based on autonomic managers that monitor these systems and their environments, analyze the captured data, plan and execute changes in the system aiming at achieving/maintaining some high-level objectives. This autonomic manager is a variation of a feedback controller inspired by the ideas from control engineering [HDPT04]. Currently, the use of various types of controllers for design and implementation of adaptive systems is growing in popularity (e.g., [BMGG09]). Therefore, the proposal presented in this chapter has wide applicability to adaptive systems in general. We explore requirements engineering for controller-based adaptive systems in more detail in Chapter 8.

The complexity of modern software-intensive systems is growing. Software is becoming truly ubiquitous, appearing in everything from cars to telephones, televisions, vending machines, etc. Together with the ever-widening use of software systems we are also experiencing an unprecedented growth in their complexity. From such a variety of heterogeneous systems the market demands seamless integration, interoperation, and adaptivity in order to achieve stakeholder goals in a changing landscape. All of these factors contribute to an enormous increase in complexity of software systems *management*. Adaptive or autonomic systems promise to move the management of this complexity from humans into the software itself, thus reducing software maintenance cost (thereby drastically diminishing the dominant cost factor in the software lifecycle), improving performance of systems, increasing customer satisfaction, etc. The distinguishing characteristic of adaptive systems is their ability to adapt to changing requirements and environment conditions, to recover from failures, and so on.

There are a number of ways for designing systems capable of adaptive behaviour. One possibility is to utilize software agents and multiagent systems (MAS). This implies the incorporation of planning/reasoning support as well as a rich set of social abilities into systems (e.g., [SKWL99]). Then, the components of adaptive systems (i.e., autonomous software agents) will be able to cooperate, coordinate their actions aimed at achieving system goals, delegate tasks to external agents, etc. to deliver required functionality. This is a powerful paradigm that can support complex tasks such as handling of dynamic and incompletely known environments, goals that are unknown at design time, etc. However, the cost of these advanced features is the enormous system complexity (including modeling, development, validation and verification, and support), the need for heavy formalization, and the lack of transparency and predictability.

We, on the other hand, propose that high-variability goal models be used as the basis for designing software that supports a space of possible behaviours, all delivering the same functionality. The main idea here is to utilize the variability implemented in software systems to support adaptation in these systems (i.e., the change in the system behaviour in response to various triggers). This approach relies on the variability-preserving transformation idea for systematically generating high-variability designs that we discussed in Chapter 5.

Here, we argue that adaptive systems must be aware of their goals, ways to achieve them, non-functional constraints on the achievement procedures as well as domain constraints. Properly augmented (with annotations) goal models are capable of capturing all of this information and thus act as the main source of knowledge as well as the intentional abstraction for the adaptive system at runtime, helping it with the selection of the appropriate behaviour alternatives.

We also discuss how various adaptive behaviours such as self-configuration, self-optimization, etc. can be supported by this approach. Not all of the alternative system behaviours can be recognized at design time and new behaviours could be identified and implemented by the system at runtime. However, this is a difficult task, which our approach minimizes the need for. Overall, the requirements-based approach for adaptive systems design presented in this chapter provides the basis for designing predictable and transparent systems. The advantages of this approach include the support for traceability of software design to requirements as well as for the exploration of alternatives and for their analysis with respect to stakeholder quality concerns. We also sketch an autonomic systems architecture that can be derived from these goal models.

## 7.2 Towards Goal-Driven Adaptive Systems

In this section, we describe how goal models can be helpful in designing autonomic application software, outline the architecture for autonomic computing (AC) systems that can be easily derived from goal models, and describe our requirements-driven approach for developing autonomic systems.

### 7.2.1 The Role of Goal Models

The building blocks of autonomic computing are architectural components called *Autonomic Elements* (AEs). An autonomic element is responsible for providing resources, delivering services, etc. Its behaviour and its relationships with other AEs are "driven by goals that its designer embedded in it" [KC03]. An AE typically consists of an autonomic manager and a set of managed elements, such as resources, components, etc. The manager must be able to monitor and control the managed elements.

An autonomic element manages itself to deliver its service in the best possible way. In order to achieve this, its autonomic manager must be armed with tools for monitoring its managed elements and the environment, for analyzing the collected data to determine whether the AE is performing as expected, for planning a new course of action if a problem is detected, and for executing these plans by, for example, tuning the parameters of its managed elements. Most importantly, these activities require the knowledge about the goal of the autonomic element, the configurations and capabilities of its managed elements, the environment of the AE, etc.

We believe that goal models can be useful in the design of autonomic computing systems in several ways. First, goal models are used to capture and refine requirements for autonomic systems. A goal model provides the starting point for the development of such a system by analyzing the environment for the system-to-be and by identifying the problems that exist in this environment as well as the needs that the system under development has to address. Thus, requirements models can be utilized as a baseline for validating software systems.

Second, goal models provide a means to represent many ways in which the objectives of the system can be met and analyze/rank these alternatives with respect to stakeholder quality concerns and other constraints, as described above. This allows for exploration and analysis of alternative system behaviours at design time, which leads to more predictable and trusted autonomic systems. It also means that if the alternatives that are initially delivered with the system perform well, there is no need for complex social interactions among autonomic elements (e.g., as implied in [KC03], where AEs are viewed as socially-capable intelligent agents). Of course, not all alternatives can be identified at design time. In an open and dynamic environment, new and better alternatives may present themselves and some of the identified and implemented alternatives may become impractical. Thus, in certain situations, new alternatives will have to be discovered and implemented by the system at runtime. However, the process of discovery, analysis, and implementation of new alternatives at runtime is complex and error-prone. By exploring the space of alternative process specifications at design time, we are minimizing the need for that difficult task.

Third, goal models provide a traceability mechanism from AC system designs to stakeholder requirements. When a change in stakeholder requirements is detected at runtime (e.g., by using the approach in [FF98]), goal models can be used to re-evaluate system behaviour alternatives with respect to the new requirements and to determine if system reconfiguration is needed. For instance, if a change in stakeholder requirements affected a particular goal in the model, it is easy to see how this goal is decomposed and which components/autonomic elements implementing the goal are in turn affected. By analyzing the goal model, it is also easy to identify how a failure to achieve some particular goal affects the overall objective of the system. At the same time, high-variability goal models can be used to visualize the currently selected system configuration along with its alternatives and to communicate suggested configuration changes to users in high-level terms.

Fourth, goal models provide a unifying intentional view of the system by relating goals assigned to individual autonomic elements to high-level system objectives and quality concerns. These high-level objectives or quality concerns serve as the common knowledge shared among the autonomic computing elements to achieve the global system optimization. This way, the system can avoid the pitfalls of missing the globally optimal configuration due to only relying on local optimizations.

### 7.2.2 A Hierarchical Autonomic Architecture

We now outline the architecture for autonomic software systems that can be derived from high-variability goal models. We envision a hierarchy of autonomic elements that is structurally similar to the goal hierarchy of the corresponding goal model. Here, leaf-level goals are to be achieved by the components of the system-to-be, by legacy systems, or by humans. Higher-level goals are used to aggregate the lower-level ones all the way up to the root goal. Additional information such as softgoal contributions and annotations is used to determine the best configuration of the system for achieving its main goal.

In the most straightforward case, a single autonomic element is responsible for the whole system. Thus, it is associated with the whole goal model and is said to achieve the root goal of the model. This has certain advantages in that all of the analysis, monitoring, etc. is done in one place, which can be helpful in achieving globally optimal performance. However, there are also potential problems with this approach. A single AE can make the system quite inflexible, hard to maintain, as well as make it impossible to reuse any part of the system.

In the other extreme case, each goal in the goal model can be associated with an autonomic element whose purpose is the achievement of that goal. The managed elements of the leaf-level autonomic elements (which correspond to leaf-goals) are then the actual components, resources, etc. Leaf-level AEs can tune and optimize these resources to deliver their objective in the best way. On the other hand, higher-level autonomic elements are not directly associated with the actual components, but are used to orchestrate the lower-level elements. The root autonomic element represents the whole software system. Thus, an AE corresponds to any subtree of the goal model. This approach has an advantage that the global high-variability design space is partitioned into autonomic elements with lower variability, thereby facilitating management and administration tasks. Also, this will improve maintainability of the system. Finally, there is the middle ground where a goal model is partitioned among a set of AEs, each of which is responsible not for a single goal, but for a goal subtree.

It remains to be seen which strategy is the best for partitioning a goal model among autonomic elements. The size of the model is an important factor here.

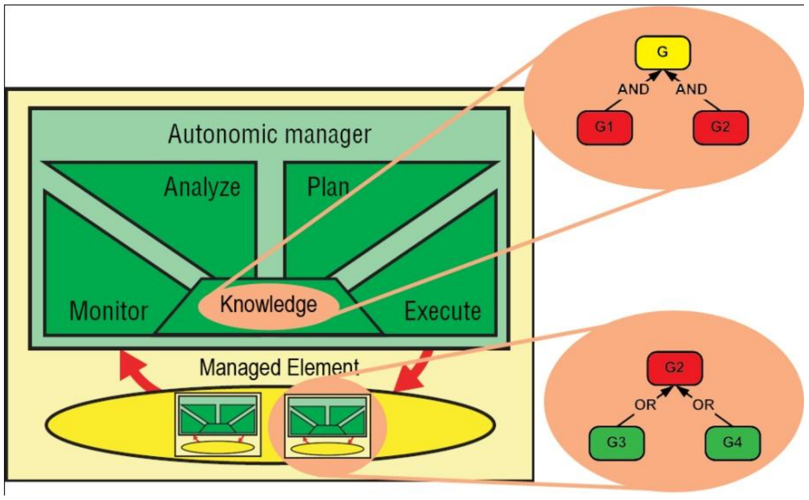


Figure 7-1. A hierarchical composition of autonomic elements.

A fragment of a properly enriched goal model will serve as the core of each AE's knowledge. For example, Figure 7-1 presents an AE, whose objective is to achieve the goal G. It has a fragment of the goal model showing the decomposition of this goal. Here, the goal G is AND-decomposed into G1 and G2, which means that the goal model identified only *one* way to achieve G. The managed elements of the AE in Figure 7-1 are themselves autonomic elements that achieve the goals G1 and G2. They have different fragments of the goal model assigned to them. For example, the AE achieving the goal G2 knows that to attain that goal it must either achieve G3 or G4 (the relevant softgoals are not shown). These goals can be in turn handled by lower-level AEs (also not shown).

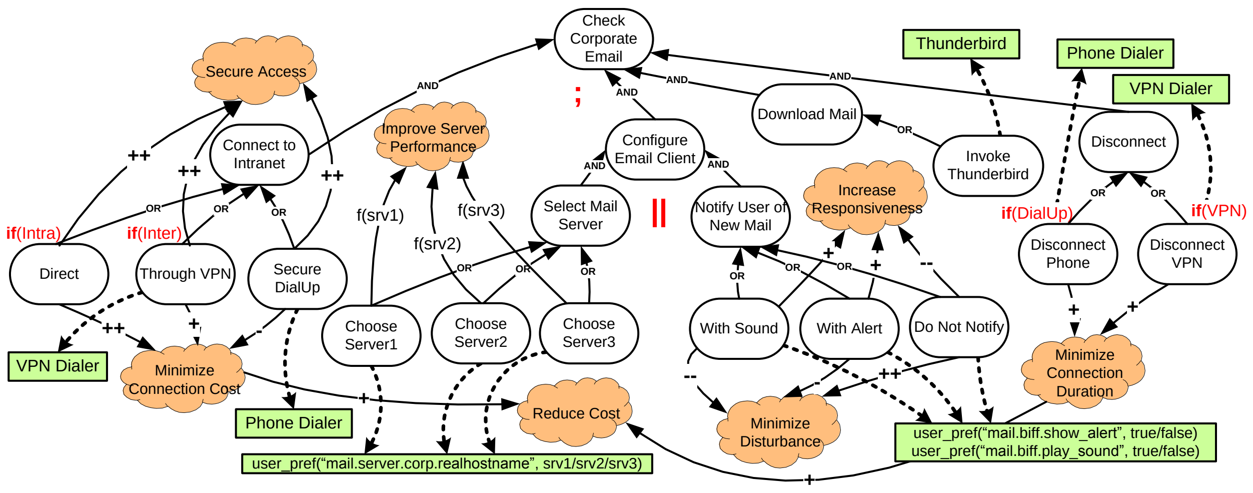


Figure 7-2. The Check Email case study.

Because of the hierarchy of AEs, it is possible to propagate high-level concerns from the root AE down to the leaf-level elements (by tuning the parameters of child AEs), thus making sure that the system achieves its objectives and addresses the quality concerns of its stakeholders. Note that each autonomic element retains the freedom to achieve its goal in the best way it can provided that it satisfies the constraints passed to it by the higher-level AE and/or the user.

### 7.2.3 An Illustrative Example

Before we describe our approach in detail, let us introduce a subset of a case study we did to evaluate it (see Figure 7-2). In the next sections we will be referring to parts of this example to illustrate the steps of the process. The example is a system designed to be used with Mozilla Thunderbird email client to periodically download email from a corporate email server, thus Check Corporate Email is its goal. First, the system needs to connect to the secure corporate *intranet*, which can be done by either connecting to it directly (through the office network), by using the virtual private network (VPN) connection, or by using a secure dial-up provider. All three ways are considered secure (note the contributions to the Secure Access softgoal), but have different costs. Rectangular shapes in the model show how leaf-level goals are implemented. For example, the achievement of Through VPN goal is delegated to an existing VPN dialer component. Then, the system configures Thunderbird to use the best email server available by selecting among the three available corporate servers. This is done by automatically changing the configuration file of Thunderbird (specifically, the parameter `mail.server.corp.realhostname`). Also, depending on whether the user prefers not to be disturbed or, conversely, prefers to be very responsive, the system configures Thunderbird to display a visual alert, play a sound, or do nothing when new mail arrives. After that, the system invokes Thunderbird and later disconnects from the intranet to reduce connection costs. As can be seen, the example system delivers its functionality by integrating and appropriately configuring existing components.

## 7.3 The Approach

In our approach for the development of autonomic software, we take high-variability requirements-level goal models as a starting point. They are used to capture the needs for the new system, both functional and non-functional and the alternatives that exist in the problem domain for meeting those needs, as well as to do the initial analysis of the alternatives with respect to the important quality criteria modeled as softgoals.

A lot of research in the Autonomic Computing area was devoted to methods and techniques for developing AC managers that handle IT resources shared among applications. These resources are usually various kinds of servers that can be dynamically allocated to applications that require them. So, the job of these AC managers is to optimize the use of their resources, to protect them, etc. Therefore, they operate in fairly restricted environments (e.g., data centres) and their decisions are implemented in terms of a relatively small set of actions that are available in these domains. This makes the AC managers quite generic (i.e., middleware-like). Moreover, most of the activities of these managers are hidden from the applications since they are quite low-level and thus do not affect these applications in a profound way. All of these characteristics make the field of resource allocation and provisioning ripe for automation.

We, on the other hand, believe that resource allocation/provisioning is just one of the areas that can benefit from autonomic computing ideas and that these ideas can be applied to systems other than AC managers – specifically, to applications themselves. Therefore, our approach is meant to be used to introduce autonomic behaviour into application software, thus making it more flexible and robust in achieving its goals.

There are a number of ways in which autonomic application software differs from autonomic middleware. First, the autonomic functionality is application-specific, not generic. Second, the changes in the autonomic application behaviour are usually visible to and have direct effect on the user and thus might require his explicit approval and his trust. Third, the autonomic behaviour of an application system is highly influenced by the preferences and priorities of its users.

The above discussion suggests that autonomic application software requires special development methodologies that address its unique characteristics. Thus, the approach presented here that is rooted in software requirements engineering and provides a way to explicitly model and analyze alternative behaviours and how they affect user quality concerns seems a promising way for building autonomic application software.

In this approach, users (perhaps, non-technical) can be in the loop by approving the software changes proposed by the autonomic system as well as by driving them by the means of, for example, shifting priorities from one non-functional concern to another. As noted before, the approach leads to more predictable and trusted systems and thus can be used for developing mission-critical systems with autonomic behaviour where changes in the system's behaviour might have to be approved by an appropriate person before they are enacted. Goal models can help with such user interaction since they explicitly represent goal achievement alternatives as well as are able to present them in high-level terms.

Since the approach relies on the manual elicitation of high-variability goal models, it may not be suited for domains that need very large number of goals. However, once the goal model is developed, the alternatives can be enumerated and analyzed automatically. For example, [HLM03] shows that even naïve algorithms can work reasonably well on a goal model with 750 nodes and $10^6$ alternatives.

We now describe the main steps in the process in more detail.

### 7.3.1 Developing High-Variability Goal Models

The process starts by identifying the problem domain, together with the opportunities for its improvement. Specifically, we look at how the introduction of a software system can improve the situation in the domain. The *i** notation [Yu93] can be used at this stage to model stakeholders in the domain along with their needs. This early requirements stage helps us in identifying the goals that the system-to-be will have to achieve. Once the goals of the system are clear, we use goal models to capture and refine them using AND/OR. The emphasis here is on modeling the variability in the problem domain: we try to capture all the different ways the system's goals can be achieved in that domain. We again refer to the approach for high-variability goal model elicitation described in [LLYY06], which can help with this task. We refine the goals of the model until we reach the ones that can easily be achieved through developing a software component, delegating the goal to an existing component, a legacy or a COTS (Commercial Off-The-Shelf) system, or a person. Also, as we can see in the Check Email system, some goals can be achieved by appropriately configuring COTS systems.

In our example in Figure 7-2, the goal of the system is Check Corporate Email. This goal is refined into subgoals with alternative refinements (e.g., the way one can connect to the corporate intranet) represented by OR decompositions. We stopped the refinement once we identified the goals that could be achieved by the existing COTS systems such as Mozilla Thunderbird email client, a VPN dialer, or by appropriately configuring the COTS products used in the system.

Non-functional constraints are used for analyzing the alternatives and for selecting the best option for the system's behaviour. In the Check Email example, the softgoals include Improve Server Performance, Increase Responsiveness, and Minimize Disturbance. Note that the latter two have the generally opposite contributions from the alternative ways of notifying the user of new email messages: the goal Do Not Notify breaks (--) the softgoal Increase Responsiveness while making (++) Minimize Disturbance. Thus, the selection of the best notification alternative will depend on how the user prioritizes among these quality constraints. A change in such prioritization will trigger a reconfiguration of the system.

While eliciting goal models, we may also add the necessary annotations for capturing additional information. Sequential and parallel annotations are used in the Check Email model. For instance, the goal of connecting to the intranet must be achieved before the goal of downloading mail. Similarly, the two aspects of the email client configuration, namely the mail server and the type of new email alert can be done independently, thus the goal Configure Email Client is used with the parallel annotation.

## 7.3.2 Adding Formal Details

While some GORE approaches (e.g., KAOS [DLF93]) require formal specifications for all goals in goal models, in our approach it is up to the user to determine to what extent the model must be formalized. This means that if automated planning is a feature of the system, then all the goals will most likely be formally specified. Otherwise, the system specification can mostly remain informal. For example, in Figure 7-2, we only specify preconditions for goals *as needed* by using conditional annotations `if(condition)`. Specifically, in Figure 7-2 the goal Connect to Intranet is OR-decomposed into the goals Direct and Through VPN referring to the ways one can connect to a corporate intranet. The precondition for Through VPN is `Inter`, which is a boolean variable that is true whenever there is internet connectivity (since you have to have the internet connection to be able to use VPN). The precondition for the direct intranet connection is the existing intranet connectivity. Preconditions capture domain properties that must be true for alternatives to be considered for a selection. For instance, if the system has only internet (but not intranet) connectivity, then the Direct option is not available, while the VPN and dial-up options are. When multiple alternatives are available, quality criteria (in this case, Minimize Connection Cost softgoal) will be used to select the best one.

Likewise, the two alternatives for the Disconnect goal, namely Disconnect VPN and Disconnect Phone, have as preconditions the VPN and dial-up connectivity respectively. Obviously, one can disconnect a dial-up connection only if it has been previously established. Thus, the boolean variable `DialUp`, a precondition for Disconnect Phone, must capture the effect (post-condition) of the goal Secure Dial-Up. The same applies to the variable `VPN` and the goal Through VPN. Therefore, when a VPN connection is established, it will be disconnected by achieving the goal Disconnect VPN. The preconditions are *a* source of requirements for the monitoring component of the system. In Chapter 8, we talk about *awareness requirements* – another source of requirements for the adaptation infrastructure, including monitoring.

Note that preconditions discussed in this chapter can usually be viewed as domain assumptions, a kind of requirement [JMF08]. Domain assumptions are utilized in Chapter 8. There, they are used as first-class citizens, featuring explicitly as nodes in goal models, not as goal model annotations.

### 7.3.3 Specifying Softgoal Contributions

In this approach, we, as usual, rely on softgoal contributions to represent how particular alternatives affect softgoals. Many of these contributions do not change throughout the execution of the system. For instance, in Figure 7-2, the goal Do Not Notify [of incoming messages] *makes* (++) the softgoal Minimize Disturbance, while the goal [notify] With Alert hurts it. This captures the understanding that any alert is a distraction. And this is unlikely to change. On the other hand, there are situations where one would like to model softgoal contributions not as constants, but as functions. In the Check Email example, such softgoal is Improve Server Performance. Suppose that the chosen way to improve email server performance in the corporate system is to make email clients connect to servers with the lowest current workload. Since server workload, obviously, varies, to pick the server with the lowest load we must parameterize the contributions to the softgoal Improve Server Performance as, for example, done in [LLWY05]. To preserve uniformity in treating softgoals and thus to still allow the use of the previously mentioned goal reasoning algorithms, we define the function `f(srv)` (where `srv` is the name of the email server), which maps certain server workload ranges into the already discussed four contribution labels. Here, we assume that maximum server load is 999 concurrent connections. The function is defined through the sensed value `load(srv)`, the current load on the server `srv`.

$$f(srv) = \begin{cases} \text{"++"}, \text{if } \mathit{load(srv)} \leq 300 \\ \text{"+"}, \ \text{if } 300 < \mathit{load(srv)} \leq 600 \\ \text{"–"}, \ \text{if } 600 < \mathit{load(srv)} \leq 800 \\ \text{"––"}, \text{if } 800 < \mathit{load(srv)} \leq 999 \end{cases}$$

Since softgoals represent quality concerns (non-functional requirements), not all of them can be automatically determined to be satisfied or not. It is even harder is assign values to softgoals thus turning them into quantitative entities. Still, many softgoals can be *metricized* – assigned metrics that approximate those concerns. The handling of the goal Improve Server Performance above is an example of metricizing a softgoal. Similarly, a popular metric for reliability is *mean time between failure* is another example. There are many examples of such well-understood metrics that can be used to approximate profitability, reliability, performance, etc. However, not all softgoals can be metricized. For example, Convenience is a highly subjective criterion. Metricizing a softgoal turns it into a *quality constraint*.

In general, in order to metricize a softgoal one needs to come up with a *measurable* function approximating that softgoal. Additionally, based on the usual four-valued system for softgoal contribution we use in our goal models the range of the function has to be partitioned into four sub-ranges, each corresponding to the contribution value from “--“ to “++” as done in the example above.

## 7.3.4 Monitoring

For a system to exhibit autonomic behaviour, it must be able to monitor its environment as well as its own behaviour to detect changes, failures, etc. Appropriately enriched goal models described in the previous sections can help in determining what information needs to be captured and analyzed by the system.

First of all, the system must be able to monitor the achievement of its leaf-level goals. These are the goals that are assigned to the system components, or the environment of the system (i.e., legacy systems, humans, etc.) In Requirements Engineering, the latter are viewed as the system's *expectations* of its environment and so an autonomic system must monitor the achievement of these goals in order to detect if the expectations are still valid.

The monitoring can be done in various ways. If a goal is assigned to a legacy system or a component, it might be possible to query that system/component to get the status of the goal. Otherwise, sensors in the environment can be used to determine if the goal has been achieved without querying the involved component(s). For instance, in the example in Figure 7-2, after a VPN dialer has been invoked to achieve the goal [connect to intranet] Through VPN, we used a simple sensor to determine if access to the internal corporate network had been granted by `ping`-ing a known intranet server.

The achievement status of non-leaf goals can usually be deduced using the algorithm of [GMNS02] that propagates the satisfaction values of leaf-level goals up towards the root of the model. For example, if the goal Through VPN is determined to be achieved, then the goal Connect to Intranet is achieved as well by the semantics of the OR decomposition.

The environment of the system also needs to be monitored to determine if preconditions for goals (domain assumptions) are satisfied. In the Check Email example, the goal [connect to intranet] Through VPN requires internet connectivity. Again, a simple `ping`-based sensor is used to determine that. The boolean variable `Inter` used within a conditional annotation applied to Through VPN is defined with the help of this sensor. Frequently, a precondition of one goal is the achievement of another. For example, a VPN connection must exist before one can disconnect it. So, the precondition for Disconnect VPN, the Boolean variable `VPN`, is, in fact, the post-condition of Through VPN. It can be tested as described above.

Since some non-functional requirements (modeled as softgoals) can be metricized using approximating functions, to calculate the values for these functions, we need to capture the data used in their definitions. For example, to evaluate the satisfaction of the softgoal Improve Server Performance (as defined in Section 7.3.3) the Check Email system needs to monitor the current server load value `load(srv)` for all email servers. Metricizing softgoals can be seen as a way of deriving quality constraints [JMF08] – basically, measurable quality requirements – from softgoals.

As already mentioned, many softgoals are too high-level/subjective to be metricized. Thus, it is not straightforward for the system to, for example, automatically verify that a particular alternative's contribution to a softgoal is correctly captured in the goal model (e.g., that an alternative, in fact, contributes negatively to the softgoal Convenience). In these cases, the system might want to confirm with the user(s) that its current configuration meets the users' quality criteria. We talk more about generating monitoring requirements for the adaptive infrastructure in Section 8.5.

### 7.3.5 Using COTS Systems

COTS or legacy systems can be given responsibility for achieving goals. This can be done in the usual way through procedure calls, messages, etc. However, another possibility for using legacy software in autonomic systems is through goal-driven configuration [LLWY05], [YLLM05] where AEs will wrap these systems making sure that their behaviour conforms to the quality preferences of system's stakeholders. The use of Mozilla Thunderbird email client in our Check Email case study is an example of that. Here, Thunderbird is being dynamically configured to achieve the functional goal Download Mail while meeting non-functional requirements such as Improve Server Performance. This configuration approach has limitations since it depends on the richness of configuration options of legacy systems. However, many complex systems have vast possibilities for configuration yielding thousands or millions of alternatives with very different properties that can be utilized in our approach.

When applied to COTS/legacy systems, our approach can be viewed as defining the infrastructure for flexible, yet predictable integration of these systems to meet higher-level customer needs.

### 7.3.6 Goal Model-Based Autonomic Behaviour

Given a goal model characterizing various ways of achieving some root goal G, one can rank these alternatives with respect to their satisfaction of the partially ordered set of quality criteria represented in the model by softgoals. For example, in the Check Email case study, if the softgoal Minimize Disturbance is of high priority, then any alternative that uses sound notification when new mail arrives will be ranked lower than any alternative that uses the display notification. Whenever the system needs to switch from one configuration to another, it tries to select the best new alternative that achieves the objective of the system while maximizing the achievement of the set of quality constraints (softgoals).

### 7.3.6.1 Self-Configuration

In our approach, when the system is first deployed, it is configured to execute the best alternative for the given (initial) preferences over softgoals. It should continue to execute the chosen alternative until changes in the environment of the system or changes in softgoal priorities invalidate it. If this happens, the system should be reconfigured and the new best alternative must be chosen. For example, in Figure 7-2, the default means for establishing the intranet connection is the Direct connection since it, unlike the other choices, has a "make" (++) contribution to the softgoal Minimize Connection Cost. Therefore, as the user of the system keeps checking his email while being connected in the office (the precondition `Intra` always holds in this case), the Direct option will remain selected. However, if the user tries to check the corporate email from home using his own internet provider, the monitoring component will detect the internet, but not the *intranet* connectivity. Therefore, the precondition for the Direct option will not be satisfied and a reconfiguration will be needed. In this case, both of the remaining alternatives will be available since their preconditions are satisfied. The autonomic manager responsible for that part of the system will then use the goal reasoning algorithm of [SGM04] to find an alternative that achieves the goal Connect to Intranet while making the best contribution to the softgoal Minimize Connection Cost. That alternative adopts the goal Through VPN. This is an example of software reconfiguration based on a change in the environment of the system.

A similar switch from one configuration to another will happen due to the change in user priorities regarding email notification (the softgoals Increase Responsiveness and Minimize Disturbance). These changes cannot be easily detected as they are normally related to the user's mood, workload, etc. Thus, the user must be able to notify the system about such changes proactively, through the use of a GUI tool. In the case study we used a simple tool (presented in [YLLM05]) that allowed users to set priorities over softgoals for the system. Once the user's input is received, the best choice for Notify User of New Mail based on the user's new priorities is found as above with the help of a reasoning algorithm. Therefore, in our approach, both the user and the system's environment can cause self-reconfiguration.

#### 7.3.6.2 Self-Optimization

The email server configuration in Mozilla Thunderbird in our Check Email example is designed to show how self-optimization can be done in our approach (see Figure 7-2). When the system is first deployed, the values `load(srv1)` through `load(srv3)` are fetched using a simple monitoring component querying the server status database. The contribution values for the servers are then calculated and the server with the lowest workload is chosen. During the subsequent runs of the system new workload values are received and contributions to the Improve Server Performance softgoal are recalculated. If applicable, a different server is chosen. Since the formula *f* produces only four discrete values for the softgoal contributions, the system will not be able to always select the server with the lowest workload because the reasoning algorithm will not be able to distinguish among servers with relatively similar workloads and thus the same contribution labels. A finer-grained approach is, of course, possible (e.g., one use numerical softgoal contribution values).

#### 7.3.6.3 Self-Healing

A failure of a software component, COTS/legacy system, or human to achieve a goal delegated to them forces the system to search for ways to heal itself. Using one of the already mentioned goal analysis algorithms [GMNS02], the system will propagate the "denied" status of the failed leaf-level goal up the goal model to determine which higher-level goals will in turn be affected by the failure. This failure propagation can be presented to the user/administrator of the system to illustrate the severity of the problem by showing the problematic portions of the system. The "top-down" goal reasoning algorithm [SGM04] is then used to find a new system configuration that satisfies the top-level goal of the system and as many of the non-functional requirements as possible.

We will now illustrate this using the example in Figure 7-2. Obviously, a failure of any child of an AND-decomposed goal will propagate to its parent. So, in our Check Email example a failure to establish an intranet connection automatically denies the top-level goal Check Corporate Email. In this case, the model has no alternative capable of achieving the top goal.

On the other hand, all of the children of an OR-decomposed goal must fail for it to be denied. For example, in, if the goal [notify user of new email] With Sound fails, then its parent goal Notify User of New Mail can still be attained since there exist other alternatives for its achievement. From the two possibilities, With Alert and Do Not Notify, and assuming that the user prefers the softgoal Increase Responsiveness, the algorithm of [SGM04] will select With Alert as the new alternative contributes positively to that softgoal (unlike Do Not Notify).

## 7.4 Discussion

Kephart and Chess suggest that overall system self-management results from the internal self-management of its individual autonomic elements [KC03]. Moreover, in their view, autonomic elements are full-fledged intelligent agents that, when assigned individual goals, will use complex social interactions to communicate, negotiate, form alliances, etc. and ultimately deliver the objective of an autonomic system. However, deriving a set of goals and policies that, if embedded into individual autonomic elements, will guarantee certain global system properties is nontrivial. Thus, there needs to be a systematic way of capturing overall system's objectives, decomposing them into lower-level goals, and assigning those goals to AEs. This problem is not addressed in [KC03] and similar approaches. The method presented here is requirements-driven and can be used to systematically derive goals for individual AEs/agents given the overall goals of the system.

Multiagent systems promise to provide a very flexible, scalable and open platform for software applications. However, the cost of introducing agent infrastructures that rely on complex interaction protocols, planning, etc. may outweigh their benefits in the domains where, for example, well-understood performance models already exist and can be used for automated optimization of software systems. Also, analysis of multiagent systems and especially of the hard-to-predict emergent behaviour is extremely difficult and requires a high degree of model formalization, which limits accessibility of multiagent systems to developers. At the same time, there are also concerns that a fully agent-based solution may not be acceptable in certain domains such as mission critical systems, business support systems, etc. where predictability, reliability and transparency are of paramount importance. Similarly, trust can be a major issue in the acceptance of AC systems. We believe that while being less flexible (since we require all of the alternatives to be explicitly represented in models), our methodology provides the capability to analyze important process alternatives thus increasing the system's predictability and transparency while improving the users' trust in it. The maintenance of systems can also be helped by our approach. For example, new ways to achieve system goals can be introduced incrementally into the system starting with their modeling at the requirements level using goal models (e.g., as an addition of a new choice to the existing variation point), followed by the generation of the appropriate executable models and the integration of the new behaviour into the system.

In [LL06], an agent-oriented requirements engineering method is introduced that translates *i** models (which are a superset of the goal models described here) into high-level formal agent specifications that support formal representation of and reasoning about goals and knowledge of agents. That approach is similar to the one presented here in the sense that it is requirements-driven and uses a similar goal-oriented notation. However, the method of [LL06] does not emphasize the variability aspect of goal models as much as we do here. Therefore, we view the two techniques as complementary to each other. By allowing leaf-level goals in our approach to be delegated to intelligent software agents, we will help with the design of systems that support a set of previously analyzed and trusted alternatives and do not require complex multiagent infrastructures as long as one of the identified alternatives can be applied. In situations when no alternative is satisfactory, the full capabilities of intelligent software agents such as the ability to reason about their goals, to communicate with each other at a semantic level, to dynamically form teams, etc. can be invoked. We plan to work on such hybrid approach in the future.

## 7.5 Conclusion

The essential characteristic of adaptive computing systems is their ability to change their behaviour automatically in case of failures, changing environment conditions, etc. In this chapter, we outline an approach for designing autonomic computing systems based on goal models that represent *all* the ways that high-level functional and non-functional stakeholder goals can be attained. These goal models can be used as a foundation for building software that supports a space of behaviours for achieving its goals and that is able to analyze these alternatives with respect to important quality and other criteria, its own state, and its environment to determine which behaviour is the most appropriate at any given moment. For such systems, goal models provide an intentional view unifying all the system components and demonstrating how they must work together to achieve the overall system objective. Goal models also support requirements traceability thus allowing for the easy identification of parts of the system affected by changing requirements. When properly enriched with relevant design-level information, goal models can provide the core architectural, behavioural, etc. knowledge for supporting self-management. Of course, an appropriate monitoring framework as well as, perhaps, learning mechanisms need to be introduced to enable self-management. The use of our approach with intelligent software agents is also possible.

The benefits of this method also include the increase in predictability and transparency of systems as well as the users' trust in them. Unlike the approaches that proposed fully agent-based systems as the basis for self-adaptive software (e.g., [KC03]) and thus need to specify and analyze complex social behaviours of agents, to predict the emergent behaviour of such multiagent systems, and to make sure that it complies with the overall requirements, we stress the need to systematically derive system behaviour from requirements. We begin by identifying alternatives in the problem domain. Then, this variability is mapped into the variability in the solution domain by using the approach presented in Chapter 5 to generate alternative behaviours achieving the requirements, thus preserving the variability in the problem domain while maintaining the traceability between variations in the requirements and in the implementation. The criteria for selecting among the implemented alternatives are derived from the quality requirements. This leads to the adaptations executed by the autonomic system being visible and predictable, thus fostering user trust into the system.

Presented here is a vision for the requirements-driven design of autonomic software. The many behavioural alternatives achieving the same system goals in high-variability software designs can serve as the basis for the design of adaptive systems. High-variability goal models can be used at runtime to guide adaptation. However, we have not yet discussed how to elicit, refine, and implement the requirements for adaptive behaviour. What are the sources of these requirements? How can we systematically derive requirements for systems monitoring, diagnosis and adaptation? In Chapter 8, we talk about one source of adaptive behaviour: awareness requirements – requirements referring to the success or failure of other requirements. We also discuss how goals of feedback controllers – processes designed to achieve adaptation requirements – can be derived from awareness requirements and subsequently refined using goal models.

# Chapter 8
# Awareness Requirements for Adaptive Systems

**Acknowledgement**: We acknowledge the contribution of Vítor Souza to the research presented in this chapter, in particular to the OCL-based formalization of awareness requirements.

## 8.1 Introduction

In Chapter 7, we show that high-variability software designs derived from the corresponding HV goal models can be used to implement adaptive behaviour. The ability to switch from a particular behaviour to another one, which delivers the same functionality, guided by the high-variability goal model at runtime is the core of the idea. While this is the mechanism for adaptive behaviour, we have not discussed precisely what guides the adaptation, i.e., what the adaptation requirements and their origins are. In this chapter, we explore one novel class of requirements – awareness requirements – that leads to adaptive behaviour as well as the high-level process for deriving adaptation requirements from awareness requirements.

There is much and growing interest in software systems that can adapt to changes in their environment or their requirements automatically or semi-manually in order to continue to fulfill their mandate. Such adaptive systems usually consist of a system proper that delivers a required functionality, along with a monitor-diagnose-plan-execute loop that operationalizes the system's adaptivity. Indications for this growing interest can be found in recent workshops and conferences on topics such as adaptive, autonomic and autonomous software (e.g., [CLGI09], [ICAC09], [SEAM09]). Feedback loops constitute an architectural prosthetic to a system proper, introducing monitoring, diagnosis, etc. functionalities to the overall system.

We are interested in studying the requirements that lead to this feedback loop functionality. In other words, if feedback loops constitute an (architectural) solution, what are the requirements problems these solutions are intended to fulfill? The nucleus of an answer to this question can be gleamed from any description of feedback loops: "... the objective ... is to make some output, say *y*, behave in a desired way by manipulating some input, say *u* ..." [DFT90]. Suppose then that we have a requirement *r* = "supply customer" and let *s* be one of the operationalizations of *r*.

Suppose further that the "desired way" of the quote for this system is that every time it is fed another customer request it does so successfully. This means that the system somehow manages to deliver its functionality under all circumstances (e.g., even when one of the requested items is not available). Such a requirement can be expressed, roughly, as *r1* = "Every instance of requirement *r* succeeds". We can generalize on this: we could require that the system succeeds more than 95% of the time over any one-month period, or that the average time it takes to supply a customer over any one week period is no more than 2 days. The common thread in all these examples is that they define requirements about the runtime success/failure of other requirements, including qualities. We call these *self-awareness requirements.*

A related class of requirements is concerned with the truth/falsity of domain assumptions. For our example, we may have designed our goods distribution system on the domain assumption *d* = "suppliers for items we distribute are always open for any instance of *r*". Accordingly, if the availability of suppliers is an issue for our system, we may want to add yet another requirement *r2* = "*d* won't fail more than 2% of the time during any 1-month period". This is also an awareness requirement, but is concerned with the truth/falsity of domain assumptions. We might call these contextual awareness requirements.

The objective of this chapter is to study awareness requirements and propose a systematic way to fulfill them through feedback. Awareness requirements are first characterized syntactically as requirements that refer to other requirements and domain assumptions, and are represented in terms of UML classes and OCL expressions. We then propose a systematic process for generating architectural elements consisting of one/several, possibly nested feedback loops to accommodate a given set of awareness requirements. We also discuss various issues that arise from such a class of requirements and the process through which they are transformed into feedback loops.

Awareness is the state or ability to perceive, to feel, or to be conscious of events, objects or sensory patterns. Awareness is a topic of great importance within both Computer and Cognitive Sciences. In Philosophy, awareness plays an important role in several theories of consciousness. In fact, the distinction between self-awareness and contextual requirements seems to correspond to the distinction some theorists draw between higher-order awareness (the awareness we have of our own mental states) and first-order awareness (the awareness we have of the environment)

[Ros06]. In Psychology, consciousness has been studied as "self-referential behaviour". Closer to home, awareness is a major design issue in human-computer interaction (HCI) and computer supported cooperative work (CSCW). The concept in various forms is also of interest in the design of software systems (security / process / context / location / ... awareness).

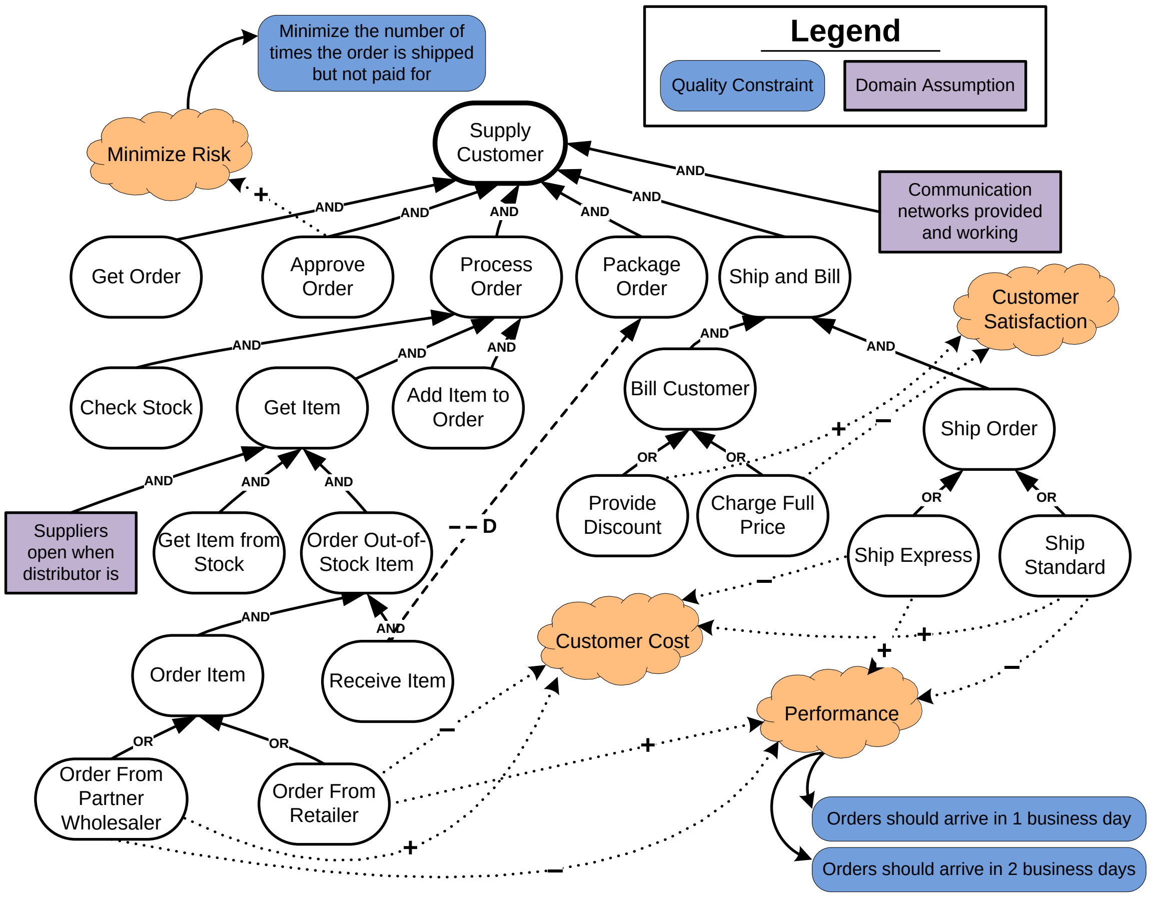


Figure 8-1. A version of the Supply Customer BP example.

## 8.2 Baseline

### 8.2.1 Goal-Oriented Requirements Engineering

For the most part, the version of the goal modeling notation that we use in this chapter is the standard one described in Section 3.1. Here, we do use domain assumptions as well as quality constraints (see Figure 8-1). For our purposes in this thesis chapter, we need to translate softgoals into quality constraints, which are perceivable and measurable entities that inhere in other entities [JMF08]. For instance, in the model of Figure 8-1, the Minimize Risk softgoal is measured by the number of times orders are shipped but not paid for, while the Performance softgoal is translated into two alternative quality constraints: orders arriving in 1 or 2 business days.

In this chapter, we use the Distributor example that was employed for the discussion of the context framework in Chapter 4 and also used in Chapter 6.

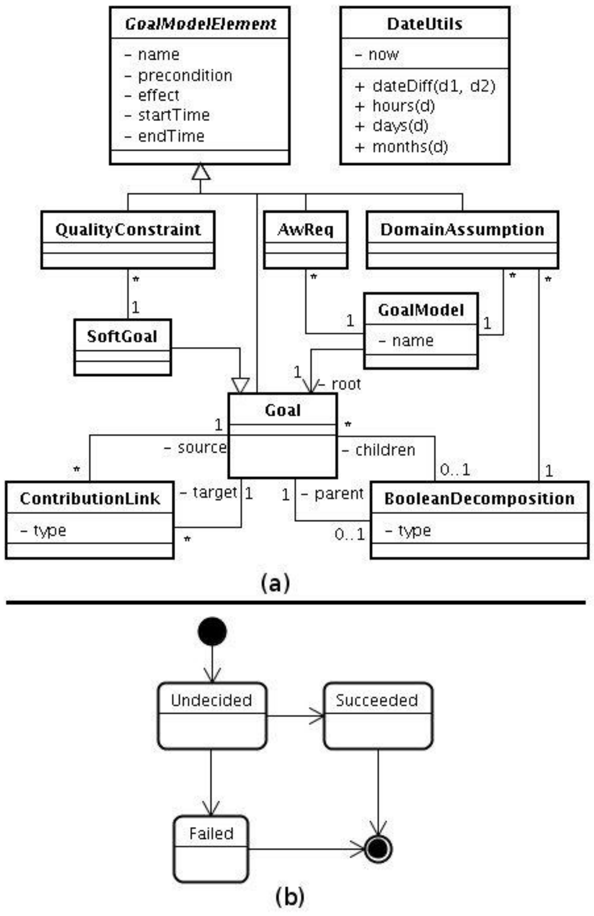


Figure 8-2.Goal meta-model for the transformation into UML classes.

## 8.2.2 UML and the Object Constraint Language

Our objective is to write requirements about the run-time success/failure of other requirements. To be able to do that, we need some way of referring to these run-time instances of the requirements. To that end, we chose the Object Constraint Language (OCL), a formal language to define expressions on UML models [OCL06].

Then, to be able to refer to the requirements we need to represent them in UML. To do that, we automatically convert a goal model into a UML class diagram using the meta-model of Figure

8-2A as the reference. Each element of the goal model is translated into a class that inherits the properties of their respective meta-class (i.e., goals extend `Goal`, etc.). Classes are associated according to the relationships in the goal model (decompositions, links, etc.). For brevity, we have manually given each class an abbreviated form of the original element's name. For instance, Supply Customer is translated into the UML class `SuppCust`, Get Order into `GetOrder`, Communication Networks Provided into `CommProv`, and so forth. In a scenario where the UML model would be generated automatically, incrementally generated IDs could be used (G1, G2, etc.). Instances of these UML classes will be created at runtime. For example, if a user accesses the functionality that implements the goal Get Order, an instance of `GetOrder` is created.

This allows us to query our goal model using OCL. For instance, if we wanted to know how many times a discount was provided, we would write the following OCL statement:

```
ProvDisc.allInstances()->size()
```

However, we mostly focus on the instances of goals, quality constraints, and domain assumptions over a period of time. For that purpose, we include the class `DateUtils` (which we abbreviate as `DU` for space constraints) in the meta-model. This class provides the current time as a property as well as the functions to calculate time differences. Thus, to see how many discounts were provided in the past 10 days, we would write:

```
ProvDisc.allInstances()->select(g | DU.days(DU.dateDiff(g.end, DU.now))
                                                            <= 10)->size()
```

But what we are interested the most is the runtime success or failure of requirements. Consider the abstract meta-class `GoalModelElement` and its statechart diagram, shown in Figure 8-2B. All goal model elements are monitorable and therefore have properties such as preconditions and effects (to check if they succeeded) as well as the start/end time of monitoring. They can be in one of three states: undecided (goal achievement has not yet finished), succeeded or failed.

Consider `l10d` as an alias to the set returned from the previous OCL expression, i.e., the executions of Provide Discount over the last 10 days. To know how many of those executions failed, we would use this OCL query:

```
l10d->select(g | g.oclInState(failed))
```

OCL provides a simple, yet powerful language for querying goal models and formalizing the kind of requirements we want to write. Section 8.3 presents awareness requirements in more detail.

## 8.3 Awareness Requirements

As mentioned in the introduction, we are interested in studying the requirements that lead to feedback loops in adaptive systems. These requirements, called awareness requirements, refer to the success of failure of other requirements at runtime.

Berry et al. [BCZ05] defined the *envelope of adaptability* as the limit to which a system can adapt itself: "since for the foreseeable future, software is not able to think and be truly intelligent and creative, the extent to which a [system] can adapt is limited by the extent to which the adaptation analyst can anticipate the domain changes to be detected and the adaptations to be performed."

For this reason, we believe that in order to completely specify a system with adaptive characteristics, adaptability requirements have to be included in specifications. We propose awareness requirements to fill this need.

### 8.3.1 Introducing Awareness Requirements

Awareness Requirements (henceforth referred to as AwReqs) are requirements that talk about the success or failure of other requirements. Using the Distributor example of the previous section we can create such a requirement that says: "shipping an order should never fail", and possibly attach a compensation action in case it does (we further discuss compensations in Section 8.5). This new awareness requirement, named A1, is formalized in OCL as:

```
def: allSO : Set = ShipOrder.allInstances()
inv A1: allSO->select(g | g.oclInState(failed))->isEmpty()
```

Some requirements may not be as critical, so we can create an awareness requirement that refers to the *success rate* of another requirement. For example, "getting an item from the stock should succeed 90% of the time" (A2):

```
def: allGI : Set = GetItem.allInstances()
def: successGI : Set = allGI->select(g | g.oclInState(succeeded))
inv A2: successGI->size() / allGI->size() >= 0.9
```

Analysts can also specify the period of time the requirement will be evaluated. For instance, “packaging an order should fail less than 5% of the time and we must have this information on a weekly basis in order to perform periodical evaluation” (AwReq A3). Other requirements might specify a maximum or minimum number of successful executions over a time period: “getting an order should succeed at least 50 times a month” (awareness requirement A4).

```
def: allPO : Set = PackOrder.allInstances()
def: weekPO : Set = allPO->select(g | DU.days(DU.dateDiff(g.end,DU.now))
                                                                  <= 7)
inv A3: weekPO->select(g | g.oclInState(failed))->size() /
                                                       allPO->size() < 0.05


def: mGO : Set = GetOrder.allInstances()->
                       select(g | DU.months(DU.dateDiff(g.end, DU.now)) < 1)
inv A4: mGO->select(g | g.oclInState(succeeded))->size() >= 50
```

Instead of a time period, some situations might call for an evaluation over a number of executions. For example: “we evaluate our partner suppliers after every 100 purchases we make, so ordering items from wholesalers should succeed 95% of the time over the last 100 executions” (awareness requirement A5):

```
def: allOW : OrderedSet = OrdWhole.allInstances()
def: s : Integer = allOW->size()
def: last100OW :
      if s > 100 then
       OrderedSet = allOW->subOrderedSet(s - 100, s)
      else
       allOW
      endif
inv A5: last100OW->select(g | g.oclInState(succeeded))->size() / s >= 0.95
```

The same awareness requirement can talk about more than one requirement (with no limit to the number of references that can be made). For example, “our discount policy mandates that charging full price succeed at least twice as much as providing discount when billing a customer" (awareness requirement A6):

```
def: allPD : Set = ProvDisc.allInstances()
def: allCF : Set = ChrgFull.allInstances()
inv A6: allPD->size() / allCF->size() <= 0.5
```

## 8.3.2 Awareness of Quality Constraints

Awareness requirements can refer not only to goals and tasks, like shown in the examples above, but also to quality constraints, thus, indirectly referencing the system's softgoals.

As discussed in Section 8.2.1, the softgoal Minimize Risk produces the QC "minimize the number of times an order is shipped but not paid for", which is a measurable metric. Over this QC we can define the awareness requirement "shipped orders should be paid for 99% of the time" (AwReq A7), thus specifying precisely what "minimize" means in this case:

```
def: allMR : Set = MinRisk.allInstances()
def: successMR : Set = allMR->select(q | q.oclInState(succeeded))
inv A7: successMR->size() / allMR->size() >= 0.99
```

The same softgoal can generate many different QCs, which in turn might be referred to by the same awareness requirement. For example, an AwReq over QCs derived from the softgoal Performance might say "70% of the orders should arrive in 1 business day, while 85% should arrive in up to 2 business days". These two awareness requirements can be formalized as:

```
def: allA1 : Set = Arrv1Day.allInstances()
def: successA1 : Set = allA1->select(q | q.oclInState(succeeded))
def: allA2 : Set = Arrv2Day.allInstances()
def: successA2 : Set = allA2->select(q | q.oclInState(succeeded))
inv A8: (successA1->size() / allA1->size() >= 0.7)
                            and (successA2->size() / allA2->size() > 0.85)
```

An awareness requirement over a QC can be viewed as a Key Performance Indicator (KPI) [CW07] related to the system at hand and thus can be used in executive dashboards to help visualize the performance of a system. Existing KPIs in an organization can be a source of AwReqs. We further discuss elicitation of awareness requirements in Section 8.4.

### 8.3.3 Awareness of Domain Assumptions

Another type of element in the goal model that can be verified for "success" (in this case, truthfulness) is the domain assumption. Since the success of the system is based on these assumptions being true, it is useful to verify if that is the case and if not, to try to mitigate the situation.

Like the other AwReq variations, awareness requirements over domain assumptions can define various levels of criticality. For instance, we might want to define an AwReq saying that "communication networks should always be working" (A9), while a less critical assumption might say "suppliers may be closed when distributor is open up to 5 times a year" (A10). These awareness requirements are formalized as:

```
inv A9: CommProv.allInstances()->select(d | d.oclInState(failed))->isEmpty()

inv A10: SuppOpen.allInstances()->
         select(d | DU.months(DU.dateDiff(d.end, DU.now)) < 12)->size() <= 5
```

### 8.3.4 Meta Awareness Requirements

An awareness requirement can also refer to another awareness requirement. A meta-AwReq is a requirement that talks about the success of another awareness requirement. This is useful, for example, if one wants to gradually apply compensations. For instance, in Section 8.3.1 we created the AwReq A6 to enforce a certain discount policy. A possible compensation for the failure of A6 is to alert the sales department head, so that he can check if there is something wrong with the orders. A meta-AwReq could say that "A6 should not fail twice in the same quarter". Its failure may trigger a notification to be sent to the financial department.

Another useful case for meta-AwReqs is to avoid executing a given compensation action too many times. For instance, the compensation in case A2 ("90% success of Get Item from Stock", Section 8.3.1) fails could be to review statistics on past orders in order to project future ones. Assume A2 is verified weekly. Since the projection procedure can be very resource-consuming, we want to avoid doing it more than once a month. Thus, we can model a meta-AwReq saying that "A2 should never fail". The compensation for this meta-AwReq would be to disable A2 for a month. This, however, does not stop us from executing A2's compensation (the projection procedure) once.

Yet another possible action in this case is to relax the AwReq: instead of disabling A2, we would lower the percentage by 10 points (from 90% to 80%) for that month. This means that if things get even worse in the next week, we should spend the resources on another projection because the first one was not good enough. These meta-AwReqs can be formalized as follows:

```
def: a6quarter : Set = A6.allInstances()->
     select(a | DU.months(DU.dateDiff(a.end, DU.now)) < 3)
inv A11: a6quarter->size() <= 1
inv A12: A2.allInstances()->select(a | a.oclInState(failed))->isEmpty()
```

With enough justification to do so, one could model an awareness requirement that refers to a meta-AwReq, which we would call a meta-meta-AwReq (or a third-level awareness requirement). There is no limit on how many levels can be created. To avoid circular references we organize goal model elements and awareness requirements in different strata, as depicted in Figure 8-3, and enforce a constraint that allows awareness requirements to only reference elements from the stratum directly below.

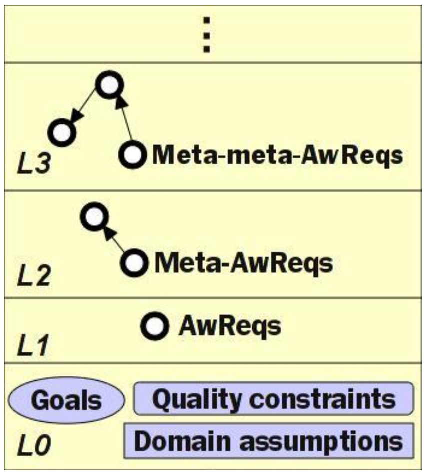


Figure 8-3. Goal model elements in their respective stratum.

## 8.3.5 Awareness Requirements Patterns

Patterns can facilitate the modeling of AwReqs. Many of them have similar structure, such as “something must succeed a certain number of times”. By defining patterns for awareness requirements we create a common vocabulary for analysts. Furthermore, patterns are used in the graphical representation of AwReqs in the goal model (presented next) and code generation tools could be provided to automatically write OCL constraints from AwReq patterns. Since the majority (if not all) awareness requirements fall into these patterns, their use can relieve requirements engineers from most of the OCL coding.

Table 8-1. Some awareness requirements patterns.

| Pattern | Meaning |
|---|---|
| NeverFail(E) | Goal model element E should never fail. |
| SuccessRate(E, r, t) | E should have at least success rate r over time t, where t is optional. |
| SuccessRateExecutions (E, r, n) | E should have at least success rate r over the latest n executions. |
| ComparableSuccessRate(E, F, x, t) | E's success rate should be at least x times as much as F's over time t, where t is optional. |
| MinSuccess(E, x, t) | E should succeed at least x times over time t. Analogous functions MinFailure, MaxSuccess and MaxFailure. |
| $P_1$ and/or $P_2$ | Pattern $P_1$ and/or pattern $P_2$ should be satisfied. |

Table 8-1 shows a non-exhaustive list of useful patterns. While the patterns in the table are rather generic, other patterns may be domain and organization-dependent. Table 8-2 shows how some of the examples given in Section 8.3.1 through Section 8.3.4 can be rewritten using patterns.

Table 8-2. Previous AwReq examples written using patterns.

| | |
|---|---|
| A1 | NeverFail(ShipOrder) |
| A2 | SuccessRate(GetItem, 90%) |
| A3 | SuccessRate(PackOrder, 95%, 1 week) |
| A5 | SuccessRateExecutions(OrdWhole, 95%, 100) |
| A6 | ComparableSuccessRate(ProvDisc, ChrgFull, 0.5) |

| | |
|---|---|
| A8 | SuccessRate(Arrv1Day,70%) or SuccessRate(Arrv2Day, 85%) |
| A11 | MaxFailure(A6, 1, 3 months) |

### 8.3.6 Graphical Representation in Goal Models

Finally, we propose that awareness requirements be represented graphically in the goal model along with other elements such as goals, softgoals, domain assumptions, and quality constraints. For that purpose, we introduce the notation shown in Figure 8-4.

Awareness requirements are represented by thick circles with arrows pointing to the element they refer to. Beside this circle icon, we write the definition of the awareness requirement using the patterns introduced earlier. Since we point to the element the AwReq refers to, the first parameter of the pattern is omitted. In case an awareness requirement does not fit a pattern, the analyst should write its name and document its OCL formalization elsewhere.

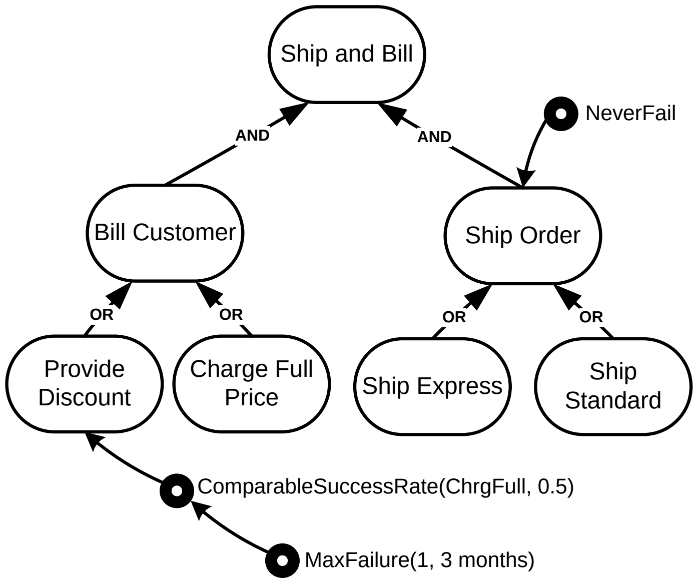


Figure 8-4. Portion of the goal model showing the graphical representation of AwReqs A1, A6 and meta-AwReq A11.

## 8.4 Eliciting Awareness Requirements

Like other types of requirements, awareness requirements must be systematically elicited. Since they refer to the success/failure of other requirements, their elicitation takes place after the basic requirements have been elicited and the goal model has been constructed. There are several common sources of awareness requirements.

One obvious source is the goals that are critical for the system-to-be to fulfill its purpose. If the aim is to create a robust and resilient system, then there have to be goals in the model that are to be achieved at a consistently high level of success. Such a subset of critical goals can be identified in the process and awareness requirements specifying the precise achievement rates that are required for these goals will be attached to them. This process can be viewed as the operationalization of high-level non-functional requirements (NFRs) such as Robustness, Dependability, etc. For example, the goal Ship Order from our case study is a critical goal for this process since the main purpose of this system is to get customers their orders. Also, government regulations and rules may require that certain goals cannot fail or be achieved at high rates. Similarly, AwReqs are applied to domain assumptions that are critical for the system.

As we have already seen in Section 8.3.2, awareness requirements can be derived from softgoals (other than Robustness, and so on). There, we discussed how the softgoal Minimize Risk is first metricized into a QC and then an AwReq is attached to that QC to state that 99% of orders need to be risk-free. This way the system is able to quantitatively evaluate at runtime whether the quality requirements are met over large numbers of process instances and make appropriate adjustments if they are not.

Qualitative softgoal contribution labels in goal models capture how goals/tasks affect NFRs, which is helpful, e.g., for the selection of most appropriate alternatives. In the absence of contribution links, AwReqs can be used to capture the fact that particular goals are important or even critical to meet NFRs and thus those goals' high rate of achievement is needed. This can be viewed as an operationalization of a contribution link. For example, the Approve Order goal in Figure 8-1 positively affects the softgoal Minimize Risk and can even be considered critical with respect to that softgoal. So, an awareness requirement, say, SuccessRate(ApproveOrder, 90%), can be added to the model to capture that fact. On the other hand, if a goal has a negative effect on an NFR, then an awareness requirement could demand a low success rate for that undesirable goal.

In Tropos [BPGM04] and other variations of the goal modeling notation such as our own approach for requirements-driven business process configuration (see Chapter 6), alternatives introduced by OR-decomposed goals are frequently evaluated with respect to certain softgoals. This is the means for selecting appropriate alternatives. The goal Ship Order in Figure 8-1 is such an example. The evaluations are qualitative and show whether alternatives contribute positively or negatively to softgoals. In our approach, softgoals are refined into QCs (which, unlike softgoals, are measurable and quantitative) and the qualitative contribution links are removed from goal models. However, the links do capture valuable information on the relative fitness of alternative ways to achieve goals. Awareness requirements can be used as a tool to make sure that "good" alternatives are still preferred over bad ones. For example, the AwReq A6 states that the full price must be charged more often than the discounted price, presumably because this is better for the Profitability NFR (omitted from the model). This way, the intuition behind softgoal contribution links is preserved. If multiple conflicting softgoals play roles in the selection of alternatives (for instance, the shipment of orders in our case study), then a number of alternative awareness requirements can be created since the selection of the best alternative will be different depending on the relative priorities of the conflicting NFRs.

One of the difficulties with AwReqs elicitation is coming up with precise specifications for the desired success rates over certain number of instances or during a certain timeframe. To ease the elicitation and maintenance we recommend a gradual elicitation, first using high-level qualitative terms such as "medium" or "high" success rate, "large" or "medium" number of instances, etc. Thus, the awareness requirement may originate as "high success rate of G over medium number of instances" before becoming SuccessRate(G, 95%, 500). Of course, the quantification of these high-level terms is dependent on the domain and on the particular awareness requirement. So, "high success rate" may be mapped to 80% in one case and to 99.99% in another. Additionally, using abstract qualitative terms in the model while separately providing the mapping to quantities helps with the maintenance of the models since the they remain intact while only the mappings are changing.

## 8.5 From Awareness Requirements to Feedback Loops

So far, we discussed what awareness requirements are and how they are elicited. Here, we present a systematic process for identifying monitoring, diagnosis, analysis, and compensation requirements based on awareness requirements.

An awareness requirement is a goal to be achieved by the adaptive system. Motivated by the recent research (e.g., [BMGG09]), which argues that feedback loops need to be first-class citizens in the SE process, in our approach, the adaptive functionality is supported by the external feedback controller processes. Then, awareness requirements are the goals of these controller processes. We can think of controllers as meta-processes designed to run the main processes, which implement the functionality of the system, in a certain way. This applies to higher-level AwReqs as well. For instance, a meta-AwReq (e.g., A11, see Table 8-2) becomes a goal of a higher-level controller designed to operate the controller for its awareness requirement (i.e., A6), thus producing a hierarchical control architecture.

Given an AwReq such as SuccessRate(Get Item, 90%), we can see how it corresponds to the elements of a standard feedback loop model with the controller's reference input being the target success rate of goal achievement, 90%. Unlike many physical systems captured with mathematical models, in this example, the process/system is specified by the goal model subtree rooted at the goal Get Item. Here, systems are controlled by modifying their goal models, e.g., by augmenting them with particular compensation goals.

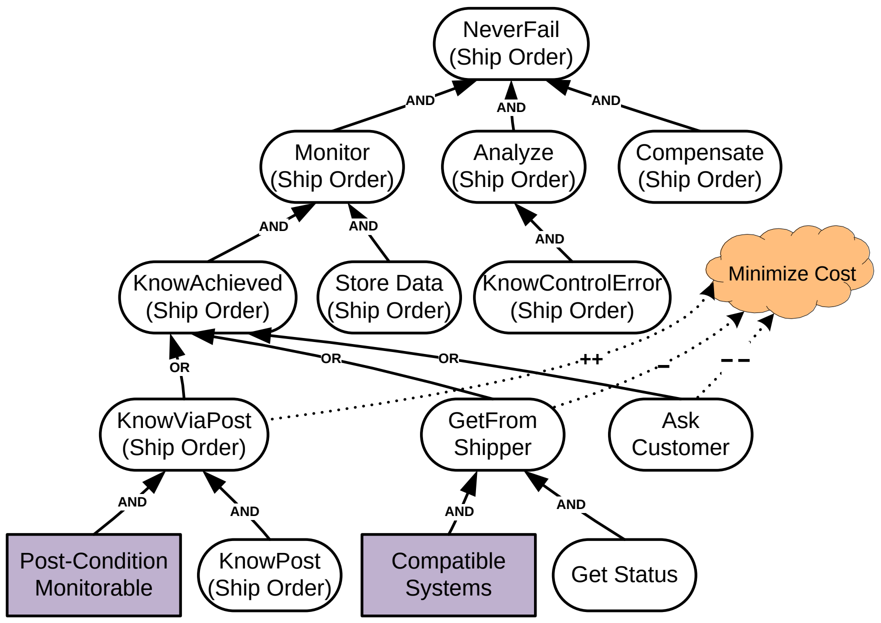


Figure 8-5. The initial goal model for the controller NeverFail(Ship Order).

When specifying the feedback controller behaviour, we are inspired by the autonomic control loop model [DDFG06] and present an awareness requirements-driven approach to elicit requirements for the main activities in feedback loops: monitoring, analysis/diagnosis, and compensation. These requirements are captured using goal models.

The goal model for the controller responsible for achieving the awareness requirement NeverFail(Ship Order) is shown in Figure 8-5. Here, the top-level goal is the awareness requirement and it is decomposed into branches that correspond to the main activities of a feedback controller.

We now present the outline of the high-level process that, given an awareness requirement, systematically derives the requirements for the feedback controller tasked with its achievement:

1. Based on the AwReq, identify the initial monitoring and analysis requirements.

2. Identify situations related to the particular awareness requirement that necessitate compensation and/or adaptations to the behaviour of the controller – adaptation situations. Optionally structure them based on their severity, origin, etc.

3. Given the adaptation situations, identify and capture the compensation/adaptive behaviour to mitigate them – refine the Compensate subgoal of the controller goal model.

4. Treat the definitions of adaptation situations from Step 2 as goals and refine the appropriate portions of the controller goal model thus eliciting additional monitoring, analysis, and diagnosis requirements (i.e., identify what needs to be monitored, analyzed, and diagnosed to determine if the system is in one of those situations).

Steps 2 through 4 are usually done iteratively. In the remainder of the section, we discuss the steps of the process in more detail.

### 8.5.1 Initial Monitoring Requirements

Given an awareness requirement, we use it to derive the initial monitoring requirements for the controller process. We first identify the goal model element(s) that it refers to: the controller must be *aware* of whether the goals are achieved, the DAs hold, and the QCs are satisfied. Figure 8-5 presents the initial refinement of the Monitor subgoal of the NeverFail(Ship Order) controller. The Monitor goal is refined by the goal KnowAchieved(Ship Order). KnowAchieved(G) is generic and models the need to be aware of whether a goal G is achieved or not. For DAs, we use the goal KnowHolds(DA), and for QCs – KnowSatisfied(QC).

In our example, there are three means of achieving KnowAchieved(Ship Order). The first, KnowPost(Ship Order), captures the need to know whether the post-condition of the goal Ship Order holds. Since we must make sure that the post-condition is monitorable (i.e., available to the system), we add the corresponding DA to the model in Figure 8-5. We use domain assumptions to represent the conditions in the environment that we assume are true – e.g., the availability of reliable sensors, the external monitoring infrastructure, certain analysis capabilities, etc.

Another way to satisfy KnowAchieved(Ship Order) is to get that information from the shipping company through the goal Get Status, provided the shipper's system is compatible with ours (the DA Compatible Systems). The third way is to ask the customer whether the order has been successfully delivered.

In general, there are many monitoring options available. Some may be predicated by domain assumptions. To evaluate and select among these options, we use softgoals. Note that these NFRs may be different from the NFRs identified for the main process! In Figure 8-5, we qualitatively evaluate the monitoring options in terms of their cost.

Alternative monitoring options for DAs and QCs can be identified and analyzed in a similar way (e.g., checking the domain assumption using a sensor, presenting a user with a customer satisfaction survey, etc.)

Awareness requirements are usually calculated over many process instances. Thus, monitored data needs to be stored for processing. Depending how the awareness requirement is to be evaluated – over what time period or over what number of goal instances – the data that needs to be stored will differ. In our Ship Order example, no timeframe is specified. It will be up to the analyst to determine how much data to store.

In addition to identifying high-level monitoring requirements, when designing controllers we are faced with numerous engineering questions such as sample rates, sensor reliability, etc. These will have to be addressed later in the design stage of the development process.

The Analyze subgoal of the controller model specifies the requirements for analyzing the monitored data and for diagnosing the system. For instance, in Figure 8-5, we identify the need to know the *control error* (the difference between the monitored output and the reference input) based on which the appropriate course of action can be taken.

### 8.5.2 Identification and Modeling of Adaptation Situations

The next step is to identify the situations requiring some form of attention from the controller – compensation or another system behaviour modification. Contexts are used to model these *adaptation situations*, which usually capture system faults, errors or failures. Positive cases (e.g., the success rate is higher than what is required by an AwReq) can be modeled as well. Contexts can be structured into hierarchies, each capturing various aspects of adaptation situations (e.g., fault severity, failure causes) at multiple levels of abstraction.

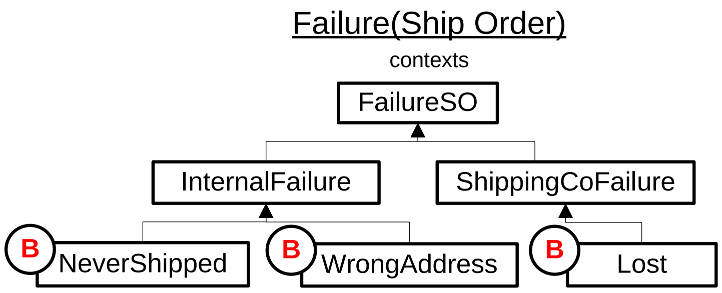


Figure 8-6. The error context hierarchy for the NeverFail(Ship Order) awareness requirement.

Figure 8-6 presents a hierarchy of adaptation situations for the NeverFail(Ship Order) controller (recall that “B” attached to a context node indicates a base or defined context). The NeverFail awareness requirement pattern allows neither for any positive adaptation situation (when the requirement is overachieved), nor for the specification of the magnitude of the error, so the top context is called *FailureSO*, meaning an instance of the goal Ship Order has failed. Using this context the analyst will specify what basic housekeeping, such as notifying appropriate people, is required for handling *any* failure to achieve the goal. To elaborate further, the context is then refined into *InternalFailure* and *ShippingCoFailure* contexts, the possible high-level sources of the failure (the names are self-explanatory), which are then refined into the possible root causes of the Ship Order failures. Overall, contexts can be used to capture various types of situations that require changes in system behaviour. These could include situations that are part of the internal context of the system, i.e., its current state or properties, its performance with respect to its targets, etc.

In many scenarios, we may not only want to identify various adaptation situations based on the severity of the control error (in order to respond *proportionally*), but also to know the *trend* in the error (if it increases or decreases) to dampen or amplify the response, thus providing the *integral* controller action. Also, the *rate* of error change can be important for selecting the correct adaptation for the process, the *derivative* action. The PID controllers providing the above three actions are very common (they can be called PI, PD, P or I controllers in the absence of the respective control actions). While PID controllers are usually used in physical systems, the use of various approaches to analyze error data and to provide appropriate control actions is applicable to computing systems as well. For example, by identifying a negative trend in the achievement of the goal Get Item from Stock, we can take a preventive action and avoid its success rate dropping to below the desired threshold of 90%.

### 8.5.3 Specifying Compensation Behaviour

Once adaptation situations have been identified, we use contextual annotations in the controller goal model to specify the adaptation/compensation behaviour that these situations demand.

Figure 8-7 shows the refinement of the Compensate goal of the NeverFail(Ship Order) controller. Here, the compensation goal Handle Failure is tagged with *FailureSO*, the top failure context. This means that the whole Handle Failure subtree applies in the case when *FailureSO* or any of its descendent contexts is active. Handle Failure is further decomposed into Deal with Customer, responsible for damage control, Assemble and Reship, the main compensation goal, Investigate, which is tagged with the *InternalFailure* context and determines who is responsible for the internal mistake, etc. Note that Deal with Customer has two alternative refinements. The selection must be made based on their evaluation with respect to important NFRs.

In addition to custom compensations (as above), there are a number of generic ones that could be employed – aimed at either re-achieving or avoiding a failed goal. They include selecting another goal achievement alternative, delegating the goal to another actor, providing more resources to goal achievement, etc. For awareness requirements involving DAs, one mitigation strategy is to acquire the DA as a goal and to explore ways to re-establish it (e.g., using planning).

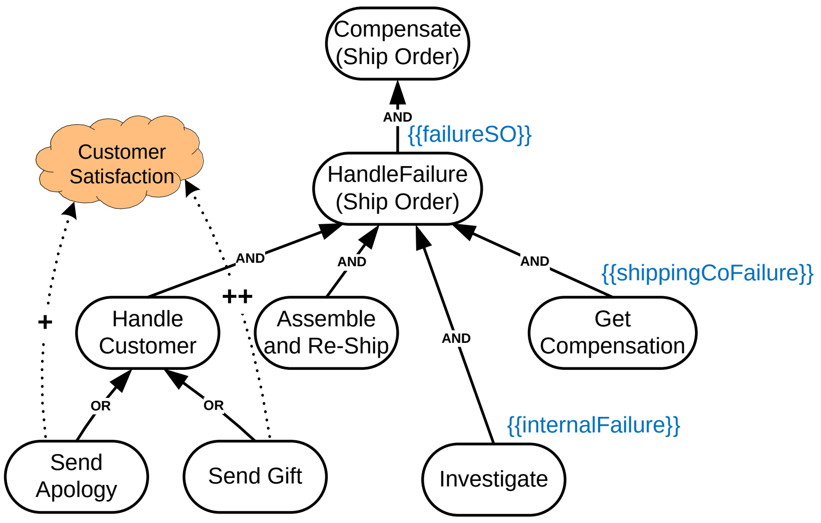


Figure 8-7. Specifying compensations.

Depending on the application needs, one can tag compensation behaviours with complex contexts such as *{ErrorIncreasing*, *LargeError*, *InternalFailure}* (i.e., the context conjunction) to specify what to do in case of a large and increasing control error due to a faulty internal component.

Quite frequently error handling and compensations are specified at the implementation level, not at the requirements level in terms of what needs to be done when a system is in a certain type of situation. What this means is that exception handling behaviour is usually not explored at the requirements level and thus may not be accessible to various stakeholders. However, the use of contexts and explicit modeling and refinement of compensation goals in our framework allows for the exposure of adaptation triggers and compensation mechanisms at the more abstract requirements level, thus explicitly linking compensations to system needs, making them amenable to analysis early on in the development process, as well as making them accessible to a wider range of stakeholders.

## 8.5.4 Expanding on Feedback Controller Requirements

The last step of the process is to expand the feedback controller requirements based on the previously identified adaptation situations (contexts). Contexts capturing various fault and adaptation conditions must be properly defined to refer to domain or error properties, system state, etc. They help identify the goals to be achieved by the controller's analysis and diagnostic component. For instance, the presence of contexts capturing positive/negative trends in the control error (e.g., *ErrorIncreasing*) gives rise to the goal Identify Error Trend, which requires that the analysis component of the controller be capable of trend analysis (thus, Identify Error Trend is a subgoal of the Analyze goal), which in turn requires storing control errors. Contexts identifying faulty components or other failure causes (e.g., *NeverShipped* and *Lost* in Figure 8-6) require the availability of the diagnosis infrastructure capable of providing such diagnoses, sometimes at multiple levels of abstraction (as in Figure 8-6), which in turn may require that more information be available to the controller (e.g., monitoring of pre-/post-conditions of many subgoals in the system). Upon the completion of this step of the process, we will have identified the monitoring, analysis, and diagnosis needs for the feedback controller derived from the adaptation situations.

The output of whole the process is a high-level adaptive architecture defined in terms of (possibly hierarchical) feedback loops together with goal models capturing their adaptation requirements.

## 8.6 Related Work

The Dagstuhl Seminar on "Software Engineering for Self-Adaptive Systems" [SESA09] discussed the state-of-the-art and challenges in this area. Challenges proposed for Requirements Engineering include a new requirements language to deal with uncertainty, systematic methods for refining this new language into an architecture, requirements reflection and traceability from requirements to implementation [CLGI09]. Andersson et al. [ALMW09] consider that "a major challenge is to accommodate a systematic engineering approach that integrates both control-loop approaches with decentralized agents inspired approaches." Brun et al. [BMGG09] notice that "while [some] research projects realized feedback systems, the actual feedback loops were hidden or abstracted. [...] With the proliferation of self-adaptive software systems it is imperative to develop theories, methods and tools around feedback loops." We believe our proposal is a starting point from the RE perspective to face these challenges.

A work with similar purpose to ours is the RELAX language by Whittle et al. [WSBC09]. RELAX aims at capturing uncertainty declaratively with modal, temporal, ordinal operators and uncertainty factors provided by the language, whose semantics are formalized in Fuzzy Branching Temporal Logic. Instead, we offer an extension to a graphical GORE language with a process aimed at coming up with high-level adaptive architecture based on feedback loops, at identifying adaptation situations and at inferring feedback controller requirements from them.

While RELAX aims at supporting unanticipated adaptations, our approach is targeting domains where predictability is important (e.g., business process management).

Proposals for self-adaptability have also come from the Tropos research group. Morandini et al. [MPP08] propose extensions to the architectural design phase of Tropos to model adaptive systems based on the Belief-Desire-Intention (BDI) model as a reference architecture. Qureshi & Perini [QP09] present a goal-based characterization of adaptive requirements that aids the analyst in modeling these kinds of requirements for a self-adaptive system. Dalpiaz et al. [DGM09] propose an architecture that, based on requirements models, adds self-reconfiguring capabilities to a system using a monitor-diagnose-compensate loop. While our proposal is similar to these works in many aspects, it differs from them in that it focus on the *requirements* for feedback loops.

Schmitz et al. [SZTJ09] uses goals to model the requirements of control systems and proposes a process to derive mathematical models from the requirements. Although our work also targets requirements for control loops, it is not restricted to control systems development and allows modeling of requirements for any adaptive system.

The approach for modeling adaptive systems proposed by Goldsby et al. [GSBC08] is in some ways close to ours. The authors base their research on the idea of the four levels of RE for dynamically adaptive systems (DAS) proposed by Berry et al. [BCZ05] and discussed by us in Section 2.4.4. They propose the modeling version of the four levels called the Levels of RE for Modeling (LoREM) that describe the modeling work done by DAS developers. The authors use *i** models for modeling these systems. There are four types of DAS developers identified: system developers, adaptation scenario developers, adaptation infrastructure developers, and the DAS research community. Thus, each of the four levels of LoREM describes the work done by one of the four categories of developers. First, at Level 1 a system developer identifies the (soft)goals of the system as well as the (soft)goals of the adaptation infrastructure. These are captured using the *i** notation, specifically the Strategic Dependency models. These models show the stakeholders (e.g., Environment Agency), the system (e.g., Flood Control System), and the adaptation infrastructure along with dependencies among them. In addition, a set of steady-state systems (i.e., $S_i$ programs in the language of [BCZ05]) for each of the possible domains for the DAS is identified. This corresponds in our approach for the identification of domain variability and capturing the effects of this variability on goal models through contexts (as discussed in Chapter 4 of this thesis). In the approach of [GSBC08], however, for each steady-state system, $R_i$, a separate *i** Strategic Rationale model is produced. For the Level 2 of RE, Goldsby et al. suggest that the current level of technology precludes DAS from doing RE at runtime, so this RE work is to be performed by adaptation scenario developers. We believe that this certainly increases the predictability and transparency of adaptive systems. Here, adaptation scenarios for the system are created. Such a scenario is an acceptable transition from a source $R_i$ to some target $R_j$. These transitions should be effected in order for the DAS to meet its requirements specified at Level 1. The adaptation models are constructed one per possible transition and capture the conditions under which the transition is to take place. At Level 3, an adaptation infrastructure developer does the RE work aimed at identifying the adaptation infrastructure capabilities that are necessary to support the previously developed adaptation scenarios. Here, the goals are to identify the monitoring, decision-making, and adaptation support that is required from the adaptation infrastructure to achieve its goals (e.g., reliability) captured at Level 1. These capabilities are modeled using the adaptation infrastructure model. This model looks similar to our goal models representing and refining the goals of feedback controllers (e.g., Figure 8-5). In [GSBC08], these models may have a goal such as Enable Adaptation as their top-level goal and refine it into monitoring, analysis, and execution subgoals. The adaptation infrastructure developer will then identify the monitoring, decision-making, and adaptation mechanisms needed to achieve the goals of the adaptation infrastructure. At Level 4, the DAS research community does RE work to determine the requirements for the major components of adaptive infrastructure (i.e., monitoring, etc.)

As we can see, there are similarities between the approach of [GSBC08] and the one presented in this chapter of the thesis. The major similarities include the use of goal models to capture requirements of adaptation infrastructures and the explicit modeling of their monitoring, analysis, etc. requirements. However, in this chapter, our focus was on *deriving* requirements for adaptive systems from a certain class of requirements, awareness requirements. Some of these awareness requirements are elicited from high-level NFRs for the system, such as robustness and dependability, which are directly assigned to the adaptation infrastructure in [GSBC08]. Additionally, as discussed in Section 9.2, awareness requirements are not the only source of adaptation requirements, with contexts and maintenance goals being some of the other sources of these. So, we envision the complete approach as taking several sources to produce complete requirements models for feedback controllers.

In this chapter, we have not concentrated on specifying adaptation requirements due to changes in context. However, we believe that the context framework of Chapter 4 can be used to capture the variations of requirements for the system under different circumstances and is thus a better alternative to using separate goal models to capture each and every possible steady-state system as done in [GSBC08]. When using the context framework, model variations for all the contexts can be captured using a single model with the possibility of a particular variation generated as needed.

In terms of modeling the requirements for feedback controllers, we refine the monitoring, analysis, compensation requirements concentrating on capturing alternatives for achieving those and propose to use goal reasoning algorithms guided by the softgoals assigned to the adaptation infrastructure to select the best choice. We also focus on modeling hierarchies of faults and precise compensation strategies based on those. Thus, we do not explicitly capture details of adaptations from one system variant (steady-state system) to another as is done at Level 2 in the approach by Goldsby et al. It remains to be seen whether our modeling approach can provide the level of detail needed to enact all of the required adaptations.

We mostly abstract from the lower-level details (e.g., the available infrastructure), but domain assumptions can be used to ground available choices into domain properties. Overall, the approach of [GSBC08] provides an excellent methodology for identifying and ordering RE activities and modeling required for the development of self-adaptive systems. It will no doubt be a source of inspiration in our future work aimed at improving and expanding the approach presented in this chapter as well as the overall variability-driven approach for adaptive systems design.

Other proposals in the literature, such as [KM09], focus at the architecture and design of self-adaptive system. While our proposal focuses on the requirements, the state-of-the-art on self-adaptive architecture will play an important role in the future steps of our research.

## 8.7 Conclusions

While in Chapter 7 we described the idea of using high-variability designs derived from high-variability goal models as a way to design and implement adaptive systems, we have not explicitly dealt with the adaptation requirements, i.e., the requirements for the adaptation infrastructure. In this chapter, we identified awareness requirements – requirements referring to the success or failure of other requirements – as *a* source of adaptation requirements. We illustrated the ideas from the portions of a business process case study developed and studied in Chapter 6. Awareness requirements are normally aggregate – they apply to many process instances or system runs (however, as we have seen with NeverFail-type awareness requirements, they can also be non-aggregate). For AwReqs to be evaluated and attained by the system, it needs to be able to monitor for whether the requirements that its awareness requirements refer to are achieved, whether the quality constraints are met, whether the environment assumptions referenced by the awareness requirements hold. Then, the system must support the analysis/diagnosis and adaptation activities aimed at changing its behaviour to achieve the awareness requirements. Awareness Requirements promote feedback loops to first-class citizens in Requirements Engineering and allow for the modeling and formalization of requirements for self-adaptive systems. The proposed systematic process aims at deriving high-level feedback loop-based architecture from AwReqs as well as at producing monitoring, analysis, and diagnosis requirements for the feedback controllers.

While we are able to explicitly model and analyze the requirements for the feedback controllers from the intentional point of view, there are a lot of details that we are not yet able to capture in our models. Take, for example, monitoring. Currently, we do not model the frequency of monitoring, at which point the monitoring should be performed, how to filter monitoring results, etc. As mentioned before, some of these details are part of the design process and cannot be dealt with adequately at the requirements level. Still, we believe that there are a lot of opportunities to increase the precision of the requirements models for feedback controllers derived from awareness requirements.

The idea of awareness requirements demands a change in perspective on how requirements of software systems are to be met. Overall, there are a number of ways to meet requirements. Some approaches are more suited for design time: their goal is to help with engineering systems that necessarily achieve their requirements with no possibility of failure built in. One such approach for developing systems is to carefully look for domain assumptions and system functionality that together satisfy the requirements, as done in the KAOS approach [DLF93]. Another way is an NFR Framework-like [CNYM00] process-oriented approach where alternatives are identified and selections among these alternatives are systematically made throughout the process based on their evaluations with respect to quality constraints, thus convincing the stakeholders that the functional/non-functional requirements of the system are met. In case of awareness requirements, the assumption is that requirements can be violated and that the mechanisms need to be in place at runtime for adjusting the behaviour of the system to restore the violated conditions, re-achieve the failed goals, etc. These mechanisms need to support monitoring of systems, environment assumptions and so on, the analysis of captured data, the diagnosis, and, finally, the adaptation of software systems either to retry the achievement of failed goals or to meet their high-level goals in some other way. Therefore, when we have this perspective on software development, in addition to the usual requirements engineering process aimed at coming up with the functional and non-functional requirements for the system, we also need to perform requirements elicitation and analysis for this runtime adaptive infrastructure.

# Chapter 9
# Conclusions and Future Work

## 9.1 Summary of Contributions

In this thesis, we presented a framework for using high-variability requirements models for the development of customizable/configurable and adaptive software systems. This research is motivated by the fact that while effort has been made in developing approaches for elicitation, modeling, and analysis of intentional variability, i.e., the various ways that system goals can be attained [LLYY06], in most goal-oriented requirements engineering approaches, this variability is used to analyze the space of alternatives and to make decisions among these alternatives using various formal or informal techniques. Then, the intentional variability is bound before a requirements specification is produced or before a software system is implemented. This results in only the "best" way of attaining system goals implemented and delivered in a software system. This limits the flexibility of the resulting systems as they are unable to easily adjust (or be adjusted) to changing conditions, varying user preferences, and environment conditions. Moreover, adapting these systems in a methodical and predictable way to improve non-functional characteristics (e.g., performance or customer satisfaction) or to avoid failures is hard.

To address these problems we suggest preserving requirements-level variability represented in goal models throughout the subsequent phases of the software development process. To this end, we proposed a variability-preserving method for generating high-variability solution-oriented models from high-variability requirements models. Goal models are first annotated to capture the additional details about the problem domain, which cannot be captured by the standard goal modeling notation, but are necessary to produce meaningful designs. Goal model patterns and their counterparts in the target notations are then identified. Emphasis here is on preserving the variability that is represented by variation points (intentional OR decompositions) in the goal models. Thus, it is crucial to determine what this variability can be translated to in the target notation and how it can be preserved there. For example, for flexible behaviour specifications using statecharts, variation points in goal models are mapped into appropriately parameterized alternative transitions. Then, the mapping process makes use of the hierarchical nature of goal models to generate a high-variability model in the target notation. Not only is the variability preserved there, but the choices made in the source goal model's variation points can be traced forward to the choices in the target notation. For instance, which alternative transition is selected in the generated statechart is determined by the selection that is made in the goal model using goal reasoning methods such as [SGM04]. We presented a number of such transformations including the generation of flexible system behaviour specifications in the statechart notation, high-variability connector-component architecture specifications, as well as feature models. Furthermore, we extensively studied the use of high-variability goal models for the design and configuration of business processes using the above approach. There, HV goal models were used to model business goals and their refinement. These models were used to generate high-variability executable BP specifications in a semi-automatic way. The specifications implemented a range of behaviours derived from goal model alternatives, which allowed them to be configured at a high level in user-oriented terms (through selecting priorities among non-functional constraints). Support infrastructure for these tasks was also developed.

Building on the above idea, we described an approach for using designs that capture many behaviour alternatives as the foundation for the development of adaptive systems. The space of possible behaviours, each achieving the same objective, makes the resulting system flexible, configurable and adaptable to changes. High-variability goal models serve as high-level intentional abstractions of the adaptive system. Due to the systematic development of this type of flexible/adaptable software and the preservation of traceability between the designs and the original goal models, the latter can be employed at runtime for reasoning about systems, for selecting alternative behaviours, etc.

In our research, we subscribe to the increasingly popular view that the functional and adaptive behaviour of systems should be treated as separate concerns [BMGG09] with the latter being assigned to some sort of a control process (usually a feedback controller), a meta-process. Therefore, in addition to the usual requirements engineering activities, adaptive systems with feedback control demand additional RE activities aimed at the elicitation and analysis of adaptation requirements. Current approaches to adaptive systems design mostly ignore the actual sources of adaptation requirements. In this thesis, we explored these sources and concentrated on a new and special class of requirements – awareness requirements. These are requirements that refer to the success or failure of other requirements. They are *a* source of adaptation requirements. In Chapter 8, we provide a framework for the elicitation and modeling of these requirements using goal models as well as their formal representation using UML class diagrams with OCL constraints for evaluation and analysis. Furthermore, we discuss the approach to derive detailed requirements for feedback controllers, specifically for monitoring, analysis/diagnosis, and adaptation. This is illustrated with portions of our business process case study. Thus, we propose a novel requirements-driven method for deriving adaptation requirement for self-managing systems that can be used in conjunction with a goal-based requirements-driven development approaches.

Our method is targeting feedback-control based adaptive systems and allows for the elicitation and refinement of the requirements for the feedback controllers responsible for the adaptive behaviour of the system. We propose a systematic elicitation of such requirements from awareness requirements (other sources of adaptation requirements will be integrated into the approach in the future work, see Section 9.2) and goal-based modeling of the feedback controller requirements that allows for explicit capture and analysis of monitoring, analysis/diagnosis, and compensation alternatives together with their evaluation with respect to quality requirements identified for the adaptation infrastructure (which may be different from the desired qualities of the main system). This makes the adaptive infrastructure more transparent by explicitly representing its goals (split into the goals for monitoring, analysis/diagnosis, and compensation components of the controller that are common to most feedback-based approaches) and refining them while supporting the modeling and analysis of alternatives. The resulting models allow representing and analysis of feedback controllers at the level of requirements. Moreover, they support the identification and explicit modeling (through contexts) of situations that require adaptations as well as the capture of the goals the adaptation infrastructure will be achieving in those adaptation situations. For example, monitoring goals and various ways of attaining them (i.e., alternative ways of collecting the necessary data) can be modeled. In our approach, adaptation triggers are represented in the model as contexts and compensation goals are captured and refined explicitly based on the types of failures and other triggers that are identified. This is in contrast with how error handling is frequently done (at the implementation level). Therefore, possible system failures and the corresponding compensations are explored at the level of requirements, in terms of system goals, which facilitates their early identification and analysis and makes them accessible to a wide range of stakeholders.

The introduction of adaptive behaviour into software systems no doubt makes them more complex and error-prone. However, the above features of our approach help with making the development of complex adaptive applications more transparent and predictable.

At the time when we came up with the ideas of using high-variability goal models as the basis for designing adaptive applications [LLMY05], which are now discussed in Chapter 7 of this thesis, Berry, Cheng, and Zhang [BCZ05] proposed a multilevel framework for dealing with requirements for self-adaptive systems. Our approach in general follows the ideas presented in [BCZ05]. For example, the first steps in our method are the elicitation and modeling of variability in the problem domain (i.e., determining alternative ways for attaining system requirements) and the systematic variability-preserving transformation of these models into solution-oriented models (i.e., the creation of the many target programs, $S_i$). Thus, the first steps in our approach correspond exactly to the activities proposed for Level 1 RE: the identification of variability in the ways requirements can be met (i.e., the intentional variability) and the systematic derivation of system behaviour alternatives that implement this variability (Chapter 5) in software. Chapter 7 of this thesis talks about the use of the alternative system behaviours (or *target programs*, as they are referred to in [BCZ05]) in adaptive systems design. We also describe how high-variability goal models can be used as part of the adaptive infrastructure for autonomic systems, which is part of Level 3 RE, and how they can be utilized by self-adaptive systems at runtime to change their behaviour if needed (Level 2 RE). Finally, Chapter 8 discusses the derivation of requirements for the feedback-based adaptation infrastructure (adaptation requirements) from awareness requirements. Elicitation of these adaptation requirements involves the identification and refinement of goals for monitoring, analysis, compensation, etc. activities. This corresponds to doing Level 3 RE.

In this thesis, we have abstracted from the implementation details, i.e., from the details on how variability is realized in software systems. We believe that many techniques such as the ones presented in Section 2.3.3 are compatible with what we propose here. This position is supported by the fact that in the business process configuration case study (Chapter 6) we used configuration files to bind variability in configurable BP implementations, while in Section 5.3.3 we show that we can generate flexible architectural representations from goal models.

We also identified the fact that modern requirements engineering approaches such as KAOS and Tropos do not provide adequate support for capturing the effects of domain variability on requirements. Variations in domain properties are an important factor in developing software systems and must be carefully taken into consideration starting from the early stages of software development. Nevertheless, goal modeling approaches assume that the environment of the system is uniform. As software systems are becoming more pervasive, the problem of the identification and analysis of the dynamic properties of these systems' environments becomes even more important. In this thesis, we propose a generic formal context framework that is used to specify for any modeling notation under which circumstances the elements of the models are present in the model instance. This is done through the use of contextual tags assigned to the elements of the model. Their definitions refer to properties of the domain. Thus, in different domain variants different contextual tags become *active* and so different portions of the model become *visible*. Tag inheritance is supported in the framework. We apply this framework to goal models to be able to represent and analyze the effects of *all* of the domain variability in a single model as well as to produce variations of the goal model for each of the contexts or combinations of contexts. Therefore, the method provides the ability to explicitly capture the requirements for particular variations of the domain thus increasing the precision of requirements models over the existing approaches. For instance, for a web-based procurement system, alternative goal refinements can be specified for various types of customers, various times of the year, etc. Moreover, the method presented in Chapter 4 supports the modeling of the effects of both external contexts (i.e., properties of the environment) as well as internal context (i.e., properties of the system itself) on requirements. The latter is exemplified by the fault hierarchies presented in Chapter 8. This approach can be used as the starting point for the requirements-driven approach for developing context-aware applications as it supports the handling of effects of contexts on requirements at the intentional level.

## 9.2 Future Work

There is a lot of work that could be done in terms of expanding, integrating, and evaluating the ideas presented in this thesis.

We are working on a larger case study to validate the approach for deriving adaptation requirements from awareness requirements (Chapter 8). An infrastructure needs to be developed to support the requirements-driven design of the controller behaviour, the evaluation of awareness requirements at runtime as well as the runtime adaptation. We plan on implementing a prototype framework with feedback loop functionality that reasons over the goal model and awareness requirements to provide self-adaptivity at runtime. Previous experience with monitoring and diagnosing based on requirements (including contexts and anti-goals) may prove useful [SM09]. Also, adopting certain implementation ideas from Tropos-inspired approaches such as [DGM09] seems promising.

One area that demands more attention in the future is the decomposition of awareness requirements. The idea is that if there is an awareness requirement applied to, for example, a goal node, then as that goal node is being decomposed into subgoals, so can be the awareness requirement – with its sub-requirements applied to the corresponding subgoals. For instance, if we require that products reach customers within 3 days of an order being placed, we can create lower-level awareness requirements asking for order processing to be done within 6 hours, order packaging within 1 day, etc. The benefit of this is that now the system is able to check *as the order is being fulfilled* whether it is on track to deliver it within the required timeframe. Having this new family of AwReqs allows the system to be *proactive* as opposed to reactive with respect to fulfilling its higher-level objective. When decomposing awareness requirements, one needs to take into account, among other things, the strength of the AwReq that is being decomposed (e.g., can a NeverFail-type requirement be decomposed into weaker ones?). Also, strata constraints need to be preserved.

The proposals discussed in this thesis should be part of a general requirements-driven framework for the elicitation and analysis of adaptation requirements for feedback control-based adaptive systems. Based on the ideas of using high-variability software design to support adaptive behaviour and of using appropriately augmented goal models as runtime abstractions of the system, the framework will support the elicitation, modeling, and refinement of adaptation requirements that come from at least the following sources:

1. Awareness requirements referring to the success or failure of functional requirements, domain assumptions, and quality constraints. The latter can be viewed as the key performance indicators that are used to evaluate the performance of software systems and business processes.

2. Maintenance goals are a useful goal type present in some goal modeling notations (e.g. [DLF93]) and approaches (e.g., [MPP08]). If the constraints captured by maintenance goals are soft (i.e., they can be temporarily violated), then the system must be monitoring for their violations and restoring these conditions when appropriate. This implies the use of a controller process together with a systematic methodology for the elicitation, refinement, and implementation of its requirements. Thus, maintenance goals are another source of control loops as well as the monitoring, analysis, diagnosis, and adaptation requirements for adaptive systems.

3. Context-awareness. In this thesis we presented a framework for representing contexts in goal models. It provides the foundation for the goal-driven modeling and analysis of requirements for context-aware systems. Such systems need to monitor their environments (for external context) as well as themselves (for internal context) and be able to change their behaviour according to the corresponding context-annotated goal models. For instance, taking an example from the Supply Customer process described in Chapter 4 and used throughout the thesis, a change in the amount of ordered goods may upgrade an order from being a medium one to being a large one. Then, certain discounts may become available to the customer that submitted the order. A context-aware system will need to be able to monitor for this change and adapt its behaviour according to the requirements for each newly identified context. This implies that there has to be an appropriate monitoring, analysis, and adaptation behaviour implemented in the system.

Additionally, all of these sources of adaptive behaviour will lead to the introduction of feedback and other types of controllers (a simpler feed-forward controller, for example, may be implemented for context-awareness). This presents a number of challenges including the problem of integrating these controllers together within a system, making sure that their adaptations are correct and consistent, and that high-level quality constraints are not violated while low-level local requirements are met. Approaches for dealing with these problems may include prioritization of adaptation requirements (and therefore the controllers implementing them) as well as clustering of the controllers that address the same concern or the same non-functional requirement (e.g., performance vs. customer satisfaction). Hierarchical controller architectures are also promising.

When gradually building and expanding on the ideas presented in this thesis as well as on other compatible approaches such as goal-driven monitoring and diagnosis [WMYM09], we will be able to develop a powerful approach for requirements-driven design of adaptive systems that will be capable of producing predictable and transparent systems.

In addition to the research aimed at the development of the overall method for elicitation, analysis, and refinement of adaptation requirements from various origins and the subsequent requirements-driven development of the controller-based adaptation infrastructure, there exist a number of opportunities for extending individual pieces of research presented in this thesis.

In Section 8.4 we mentioned that conflicting softgoals can generate conflicting quality constraints which could, in turn, be referenced by conflicting awareness requirements. Further investigation is needed to analyze the impact of these kinds of conflicts on awareness requirements.

When discussing awareness requirements elicitation (Section 8.4), we pointed out that AwReqs can originate from NFRs such as Dependability, Robustness, etc. An interesting related problem is the development of a methodology for using our approach for high-survivability systems. There, one would identify a critical subset of system goals – its core – with low tolerance for failures. Then, NeverFail or high success rate awareness requirements can be attached to this subset of goals, thus deriving the appropriate adaptation infrastructure aimed at achieving those goals in spite of various failures. Another facet for designing for survivability is the more careful analysis of domain assumptions that are required for systems to work. For instance, assumptions about power availability or network connectivity are usually implied, but not explicitly represented. However, for the development of highly survivable systems one would need to explicitly specify the behaviour of the system in the situations where those and other critical assumptions are not valid. The context framework applied to requirements models would help in identifying how system goals can be achieved in exactly those problematic situations.

At runtime, feedback loops will monitor the success and failure of goals, domain assumptions, and quality constraints, lower-strata awareness requirements and apply compensations in order to mitigate possible failures. However, goals achieved as part of the compensation/adaptation process can themselves fail, thus requiring further compensations. Solutions to this kind of problem need to be delineated. We should also take into consideration important control theory issues such as stability, accuracy, robustness, etc.

While we discussed the modeling of monitoring requirements for adaptive systems (Section 8.5.1), we mainly focused on *what* needed to be monitored. Other important issues that need to be considered include, for example, the frequency of monitoring as well as *when* to monitor (obviously, it makes sense to check if a post-condition of a goal holds after an attempt to achieve that goal has been made). Similar questions need to be answered in the case of monitoring for the achievement of QCs. Normally, softgoals and thus QCs have a clear scope: they are applied to the whole system/process (e.g., [Minimize] Cost) or to some portion of it (e.g., Order assembled within 1 day). This gives indication as to when they need to be evaluated. Also, at this high level of abstraction we avoided the issues of the availability, precision, and reliability of sensors. These problems need to be addressed in the future.

Awareness requirements, as described in Chapter 8 of this thesis, are applied to all instances of the specified goals, domain assumptions or quality constraints. For example, SuccessRate(Schedule Meeting, 90%) is applied to any instance of the goal Schedule Meeting. In many situations this may be too coarse-grained and inflexible. Goal parameters can help increase precision of awareness requirements by allowing us to apply them not only to a goal in general, but to certain subsets of that goal's instances.

For example, suppose that the goal Schedule Meeting will have the following parameters:

- `Initiator: Employee`, the employee who initiates the meeting;
- `Participants`: **`set`**`(Employee)`, the intended participants;
- `Loc: Location`, the location of the meeting;
- `Time: Date`, the date/time of the meeting.

Thus, the goal becomes MeetingScheduled(Initiator, Participants, Loc, Time). This should be accompanied by the appropriate domain model. For example, we can use UML class diagrams to model the `Employee` class with the subclass `Manager`, as well as the class `Location` with two subclasses, `InternalLocation` and `ExternalLocation`. Then, we can increase the precision of awareness requirements by constraining the instances of Meeting Scheduled as needed:

1. If we want to maintain a 90% success rate for scheduling meetings where the initiator is a manager, we can specify that:

   ```
   ∀ Initiator:Manager SuccessRate(
               ScheduleMeeting(Initiator, Participants, Loc, Time), 90%).
   ```

   Here, the rest of the parameters are not bound, so the AwReq will be restricted to the instances of the goal when the meetings are initiated by any manager.

2. If we want to state that the success rate for scheduling meetings to take place in the main office building ought to be 90%, but for the meetings taking place outside the target is just 70%, we state that:

   ```
   ∀ Loc:InternalLocation SuccessRate(
               ScheduleMeeting(Initiator, Participants, Loc, Time), 90%),
   ```

   and

   ```
   ∀ Loc:ExtrenalLocation SuccessRate(
               ScheduleMeeting(Initiator, Participants, Loc, Time), 70%).
   ```

We can also refer to the size of the set of intended participants and require a higher (or allow for a lower) success rate for larger meetings. Or we can state that meetings initiated by the CEO of the company have to have the success rate of 100%.

In addition, this opens up a new possibility for *relaxing* awareness requirements in case the system is unable to satisfy them. Previously, the main relaxation option was to reduce the success rate for achieving the goal (say, from 90% to 80%). Now we have much more flexibility in how we can relax the awareness requirements. For instance, instead of reducing the desired success rate across the board, we can reduce it for:

- all the meetings except the ones initiated by the management or
- all the meetings taking place outside of the main office or
- all the meetings taking place during lunch,
- etc.

This allows us to dynamically reduce or adjust success rates or other types of awareness requirements for most goal instances while maintaining the high success rate requirements for the instances of system goals that are deemed critical or of high importance. Therefore, we are integrating goal parameters into the approach presented in Chapter 8 and exploring the new opportunities in terms of the specification of adaptive and compensating behaviour that this modification offers. In addition, AwReqs can be context-dependent, i.e. they can refer to the success or failure of requirements and assumptions in certain contexts. For instance, taking an example from Chapter 8, a discount policy could be differentiated between regular and important customers, or the AwReq A10 over domain assumption SuppOpen can be made stricter during periods where there is a lot of demand for distribution services.

For the context framework, the major issue is the implementation. We have identified the ConceptBase implementation of the Telos language [JJM09] as a promising possibility for implementing our context framework. There is a formalization of the *i** modeling framework (which can be viewed as a superset of the basic goal modeling notation) in ConceptBase, which has already been used in a variety of projects (e.g., on mapping requirements models into systems control software [SZTJ09]). The extensibility o the language will allow for a simple extension to support contextual tag assignment to the nodes of goal models and the extensive querying facilities of the language together with the tool support will help in retrieving portions of the context-enriched goal models for particular contexts. Given an implementation, the approach can be developed further to support runtime context-driven adaptation. Also, for adaptive systems design, we need to consider advanced context issues such as context volatility, scope, monitoring, etc., some of which were identified in [HI04].

Another area where we envision a promising use of contexts and our context framework is the modeling of internal context of systems and its effect on the requirements of the system. We have already started using internal context hierarchies in our awareness requirements approach presented in Chapter 8. There, contexts were used to represent various kinds of failures in the adaptive system, from the more general failure to achieve the goal Ship Order to the more concrete one, representing the fact that the order was lost due to the error on the part of the shipping company. When they are properly defined based on the monitored data about the system state, system metrics and so on, these contexts provide the modeler with the ability to explicitly capture the goals of the system (or the feedback controller) in situations similar to the ones described above. For instance, various compensation goals can be specified to deal with different kinds of failures. Furthermore, contexts can be used to model the performance of the system with respect to the desired quality constraints (e.g., KPIs). Here, we can capture the fact that a business process is not achieving the desired performance target and be able to explicitly specify in a goal model how this situation affects the system and/or the controller goals. For instance, in case the system is falling behind in terms of certain quality constraints, goals that are not considered essential to the process can be dropped to improve performance, reduce cost, etc. while the system aims at delivering the most important services while improving its performance targets.

A lot remains to be done in terms of developing methods and techniques for using goal models to help with the users' involvement in the design and operation of adaptive systems. As already mentioned in Section 2.4.5, goal models are high-level abstractions that help with requirements elicitation by modeling the intentions behind the systems – through capturing goals and many ways to achieve them (i.e., intentional variability). Moreover, goal models promise to be a great vehicle for the visualization of the state of adaptive systems. They can be used to present the currently selected configuration by showing the choices made in variation points, as well as to display the location of failed goals and the effect these failures have on higher-level goals and thus on the overall system objectives. Alternatives represented in goal models can be seen as possible mitigations. It remains part of the future work to develop techniques to help with such visualizations as well as methods aimed at soliciting input from users during the operation of adaptive systems developed using the approach presented in this thesis.

A drawback of the approach presented here is the fact that alternatives for achieving system goals and the corresponding behaviours are explicitly modeled and analyzed. While, as already mentioned, this helps transparency and predictability, in some domains it may be infeasible to capture all the alternatives explicitly. For these domains, one may want to be able to specify alternatives declaratively, for instance, by stating the goals and constraints on their achievement and allowing planning and reasoning components at runtime to identify the best way to achieve those goals. While that approach will provide extra flexibility and power and will be very useful for certain categories of tasks and in certain domains, the transparency and visibility of solutions will be diminished. Still, declarative behaviour specifications can probably be used in adaptive middleware where behavioural changes are frequently not apparent to the user.

In the thesis, we have been advocating for the preservation of requirements variability throughout the software development process. We acknowledge that implementing many alternative behaviours achieving the same system goals demands a lot of resources. The flexibility of high-variability designs will always have to be balanced with the cost, time, and other resources devoted to systems development. Thus, software organizations will be encouraged to analyze requirements variability in terms of the feasibility, cost to implement and run, applicability, etc. of behaviour alternatives that are to be derived and implemented from these requirements.  The estimation of the overhead of designing for variability remains part of the future work.

Agent-based approaches to adaptive systems appear to be useful for supporting self-organization and can help with integrating multiple feedback controllers within adaptive systems. Part of the appeal of agent technologies is the agents' social ability. In certain domains this may play a crucial role in supporting the required level of dynamism and adaptivity through, for example, dynamic team formation and a comprehensive ability to delegate goals. While powerful, these technologies add enormous complexity to systems in terms of specification and implementation effort as well as in terms of validation and verification. Nevertheless, we believe that integration of the approach presented in this thesis with powerful agent and other AI-based approaches is a promising avenue for research [LL10].

It seems that our approach can support certain types of software evolution. For instance, a change in softgoal priorities can be handled without changing the system: the goal reasoning algorithms used for the selection of appropriate alternatives will be able to handle that (as shown in our BPM case study in Chapter 6). While an addition of new softgoals will not change the behaviour of the system, the adaptation infrastructure will need to be updated since it needs to monitor for the achievement of the quality constraints to be derived from the new softgoals, analyze the performance of the system with respect to these constraints, and possibly perform adaptations. If the system is modified to address a particular concern or to make it run in a particular environment, these new conditions can be represented as contexts and the effects of these context on the requirements can be captured in a context-enriched goal model as described in Chapter 4. The new behaviours, which specify the execution of the system in the new contexts, will be identified and will need to be implemented in addition to already existing behaviours. It remains an interesting avenue of research to develop methods for the use of our approach to support software evolution.

Moreover, we believe that iterative software development can also be supported by our framework. For instance, certain changes to the system can be quite easily handled by the approach (as described above for the case of software evolution). Then, the context framework can be adapted to help with version control of goal models. For example, the base goal model can be annotated with the *version1* contextual tag. For a new system iteration, the context *version2* can be created and declared a descendant of *version1*, thus automatically inheriting all the model elements from the first version of the system. The second system iteration can introduce new goals (tagged by *version2*), remove certain other goals by making use of non-monotonic inheritance supported by the context framework, etc. It remains to be seen how well our approach suits iterative development of goal models as well as software systems derived from these goal models.

Along a different direction, we would like to apply the context framework to *i** or Tropos models, thus supporting social modeling with actors, dependencies, etc. This will open up opportunities to use the framework in new areas such as security. For example, for the analysis of security, privacy and other aspects of information systems, a single context-enriched model can represent situations where an actor is benevolent versus malicious, with distinct contextual tags used applied in these two situations and the possibility of automatically generating the multi-stakeholder models for each of these cases. Related to the above discussion is the fact that in this thesis, we have overlooked one important type of variability related to the social landscape of the system: the variability in the way goals are delegated to actors in the system and its environment (this is also related to the problem of determining the system boundary). Since we use goal models without explicit representation of actors, the analysis of goal delegation/assignment is difficult. While certain types of analyses can be simulated in our models by, for example, refining leaf-level goals into subgoals, one for each possible actor assignment, the analysis of complex social networks with transitive responsibility delegations and other advanced features is still impossible. We plan to explore the opportunity to integrate our ideas with of approaches that focus on the analysis socio-technical systems and social networks, such as Bryl et al. [BGM09], to add support for this important type of variability.

# Bibliography

[ACMS06] S. Agarwala, Y. Chen, D. Milojicic, K. Schwan. QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems. In Proc. *3rd International Conference on Autonomic Computing (ICAC 2006)*, Dublin, Ireland, 2006.

[AHW03] N. Arshad, D. Heimbigner, A. Wolf. Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. In Proc. IEEE Conference on Tools with Artificial Intelligence, 2003, pp. 39–46.

[ALMW09] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Modeling Dimensions of Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems*. LNCS Vol. 5525, pp. 27–47, Springer-Verlag, Berlin Heidelberg, 2009.

[AMP94] A. Anton, W. McCracken, C. Potts. Goal Decomposition and Scenario Analysis in Business Process Reengineering. In Proc. *6th Conference On Advanced Information Systems Engineering (CAiSE'94)*, Utrecht, Holland, June 1994.

[Ant06] R. Anthony. Emergent Graph Colouring. In Proc. *Workshop on Engineering Emergence for Autonomic Systems*, 2006, pp. 2–13.

[BC96] R. Buhr, R. Casselman. Use Case Maps for Object-Oriented Systems. Prentice Hall, 1996.

[BCDW04] J. Bradbury, J. Cordy, J. Ringel, M. Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. In Proc. *1st ACM SIGSOFT Workshop on Self-Managed Systems*, Newport Beach, CA, USA, 2004, pp. 28–33.

[BCMN03] F. Baader, D. Calvanese, D. McGuinnes, D. Nardi, P. Patel-Schneider (Eds.). The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, 2003.

[BCZ05] D. M. Berry, B.H.C. Cheng, and J. Zhang. The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems. In Proc. *REFSQ '05*. Porto, Portugal, 2005, pp. 95–100.

[BDKW03] C. Boutilier, R. Das, J. Kephart, W. Walsh. Towards Cooperative Negotiation for Decentralized Resource Allocation in Autonomic Computing Systems. In Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico, 2003, pp. 1458-1459.

[BGGB03] P. Bouquet, C. Ghidini, F. Giunchiglia, E. Blanzieri. Theories and Uses of Context in Knowledge Representation and Reasoning. *Journal of Pragmatics*, 35(3):455–484, 2003.

[BGM09] V. Bryl, P. Giorgini, J. Mylopoulos. Designing Socio-Technical Systems: from Stateholder Goals to Social Networks. *Requirements Engineering*, 14(1):47–70, 2009.

[BFKL99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T.Widen, J.-M. DeBaud. PuLSE: A methodology to develop software product lines. In Proc. *5th Symposium on Software Reusability (SSR'99)*, Los Angeles, USA, 1999, pp. 122–131.

[BL85] R. Brachman, H. Levesque (Eds.). Readings in Knowledge Representation. Morgan Kaufmann, 1985.

[BLP05] S. Buhne, K. Lauenroth, K. Pohl. Modelling Requirements Variability across Product Lines. In Proc. *13th IEEE International Requirements Engineering Conference*, Paris, France, Aug 29-Sep 2, 2005, pp. 41–50.

[BMGG09] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Muller, M. Pezze, and M. Shaw. Engineering Self-Adaptive Systems through Feedback Loops. In *Software Engineering for Self-Adaptive Systems*. LNCS Vol. 5525, pp. 48–70, Springer-Verlag, Berlin Heidelberg, 2009.

[BP01] M. Brandozzi, D.E. Perry. Transforming Goal-Oriented Requirements Specifications into Architectural Prescriptions. In Proc. *Software Requirements to Architectures Workshop (STRAW) at ICSE 2001*, Toronto, Canada, 2001.

[BPEL07] OASIS: Web Services Business Process Execution Language Version 2.0 Primer (Draft), 2007. Available at: www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel

[BPGM04] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[BPMN04] S. White. Business Process Modeling Notation (BPMN) Version 1.0. Business Process Management Initiative, BPMI.org, May 2004.

[BPP02] P. Brezillon, L. Pasquier, J-C. Pomerol. Reasoning with Contextual Graphs. *European Journal of Operational Research*, 136(2):290–298, 2002.

[Bre99] P. Brezillon. Context in Problem Solving: A Survey. The Knowledge Engineering Review, 14(1):1–34, 1999.

[BSPM02] J. Bigus, D. Schlosnagle, J. Pilgrim, W. Mills, Y. Diao. Able: A Toolkit for Building Multiagent Autonomic Systems. *IBM Systems Journal*, 41(3):350–371, 2002.

[CCMP06] C. Cappiello, M. Comuzzi, E. Mussi, B. Pernici. Context Management for Adaptive Information Systems. *Electronic Notes in Theoretical Comp Science*, 146(1):69–84, 2006.

[CDFM07] S. Ceri, F. Daniel, F. Facca, M. Matera. Model-Driven Engineering of Active Context-awareness. *World Wide Web*, 10(4):387–413, 2007.

[CE00] K. Czarnecki and U. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading, 2000.

[CH98] P. Checkland, S. Holwell. Information, Systems and Information Systems – Making Sense of the Field. Wiley, 1998.

[CL04] L.M. Cysneiros, J.C.S.P. Leite. Non-Functional Requirements: From Elicitation to Conceptual Models. *IEEE Transactions on Software Engineering*. 30(5):328–350, 2004.

[CLGI09] B.H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software*

*Engineering for Self-Adaptive Systems*, LNCS Vol. 5525, pp. 1–26, Springer-Verlag, Berlin Heidelberg, 2009.

[CN01] P. Clements, L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, 2001.

[CNYM00] L. Chung, B. Nixon, E. Yu, J. Mylopoulos. Non-Functional Requirements in Software Engineering. Kluwer, 2000.

[CW07] S. Corea and A. Watters. Challenges in Business Performance Measurement: The Case of a Corporate IT Function. In Proc. *5th International Conference on Business Process Management (BPM 2007)*, Brisbane, Australia, Sep 24-28, 2007. G. Alonso, P. Dadam, and M. Rosemann (Eds.), LNCS Vol. 4714, pp. 16–31, Springer-Verlag, Berlin Heidelberg, 2007.

[DAH05] M. Dumas, W. van der Aalst, A. ter Hofstede. Process-Aware Information Systems: Bridging People and Software through Process Technology. Wiley, 2007.

[DDFG06] S. Dobson, S. Denazis, A. Fernandez, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, F. Zambonelli. A survey of Autonomic Communications. *ACM Transactions on Autonomous Adaptive Systems (TAAS)*, 1(2):223–259, 2006.

[DeM78] T. DeMarco. Structured Analysis and System Specification. Yourdon Press, 1978.

[DFT90] J. Doyle, B. Francis, and A. Tannenbaum. Feedback Control Theory. McMillan, 1990.

[DGM09] F. Dalpiaz, P. Giorgini, and J. Mylopoulos. An Architecture for Requirements-Driven Self-Reconfiguration. In Proc. *21st International Conference on Advanced Information Systems Engineering (CAiSE 2009)*, LNCS Vol. 5565, pp. 246–260, Springer-Verlag, Berlin Heidelberg, 2009.

[DLF93] A. Dardenne, A. van Lamsweerde and S. Fickas. Goal-Directed Requirements Acquisitions. *Science of Computer Programming*, 20:3–50, 1993.

[Eas93] S. M. Easterbrook. Domain Modelling with Hierarchies of Alternative Viewpoints. In Proc. *1st IEEE International Symposium on Requirements Engineering (RE'93)*, San Diego, January 1993.

[Eib05] A. Eiben. Evolutionary Computing and Autonomic Computing: Shared Problems, Shared Solutions? In O. Babaoglu, M. Jelasity, et al. (Eds.) *Self-Star Properties in Complex Information Systems*, LNCS Vol. 3460, Springer-Verlag, Berlin Heidelberg, 2005.

[Fau01] S.R. Faulk. Product-line Requirements Specification (PRS): An Approach and a Case Study. In Proc. *5th International Symposium on Requirements Engineering (RE'01)*, Toronto, Canada, August 2001, pp. 48–55.

[FFLP98] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard. Reconciling System Requirements and Runtime Behavior. In Proc. *9th International Workshop on Software Specification and Design*, Ise-Shima, Japan, 1998, pp. 50–59.

[FHSE06] J. Flock, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, E. Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, 2006.

[Fil68] C.J. Fillmore. The Case for Case. In Bach and Harms (Ed.), Universals in Linguistic Theory. Holt, Rinehart, and Winston, New York, 1968, pp. 1–88.

[FPF98] W. Frakes, R. Pieto-Diaz, C. Fox. DARE: Domain Analysis and Reuse Environment. *Annals of Software Engineering*, 5(1):125–141, 1998.

[FPMT01] A. Fuxman, M. Pistore, J. Mylopoulos, P. Traverso. Model Checking Early Requirements Specifications in Tropos. In Proc. *5th International Symposium on Requirements Engineering (RE'01)*, Toronto, Canada, August 2001.

[FUB06] D. Fischbein, S. Uchitel, V. Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In Proc. *ISSTA 2006 workshop on Role of software architecture for testing and analysis (ROSATEA'06),* New York, USA, 2006, pp. 39–48.

[GCHS04] D. Garlan, S.W. Cheng, A. Huang, B. Scmerl, P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[GCS03] D. Garlan, S.W. Cheng, B. Schmerl. Increasing System Dependability through Architecture-Based Self-Repair. In R. de Lemos, C. Gacek, A. Romanovsky (Eds.) *Architecting Dependable Systems*, LNCS Vol. 2677, Springer-Verlag, Berlin Heidelberg, 2003.

[GFD98] M. L. Griss, J. Favaro, M. d'Alessandro. Integrating feature modeling with the RSEB. In Proc *5th International Conference on Software Reuse (ICSR '98)*, Victoria, BC, Canada, June 1998, pp. 76–85.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[GMNS02] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, R. Sebastiani. Reasoning with Goal Models. In Proc. *21st International Conference on Conceptual Modeling (ER2002)*, Tampere, Finland.

[Gom00] H. Gomaa. Object-Oriented Analysis and Modeling for Families of Systems with UML. In Proc. *6th International Conference on Software Reuse (ICSR)*, London, UK, 2000, pp. 89–99.

[GS02] D. Garlan, B. Schmerl. Model-Based Adaptation for Self-Healing Systems. In Proc. *1st Workshop on Self-Healing Systems,* Charleston, USA, 2002, pp. 27–32.

[GSBC08] H. Goldsby, P. Sawyer, N. Bencomo, B.H.C. Cheng, D. Hughes. Goal-based Modeling of Dynamically Adaptive System Requirements. In Proc. *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2008),* Belfast, UK, Mar 31-Apr 4, 2008.

[HCS05] D. Hong, D. Chiu, V. Shen. Requirements Elicitation for the Design of Context-aware Applications in a Ubiquitous Environment. In Proc. *ICEC 2005,* Xian, China, August 15-17, 2005.

[HDPT04] J.L. Hellerstein, Y. Dao, S. Parekh, D.M. Tilbury. Feedback Control of Computing Systems. Wiley, Chichester, UK, 2004.

[HDS06] J.H. Hayes, A. Dekhtyar, S.K. Sundaram. Advancing Candidate Link Generation for Requirements Tracing: the Study of Methods. *IEEE Transactions on Software Engineering*, 32(1): 4–19, 2006.

[HI04] K. Henricksen, J. Indulska. A Software Engineering Framework for Context-Aware Pervasive Computing. In Proc. *PERCOM'04*, Orlando, FL, USA, March 2004.

[HLM03] B. Hui, S. Liaskos, and J. Mylopoulos. Requirements Analysis for Customizable Software: Goals-Skills-Preferences Framework. In Proc. *11th IEEE International Requirements Engineering Conference (RE'03)*, Monterrey, CA, USA, September 2003, pp. 117–126.

[HM08] M. Huebscher, J. McCann. A Survey of Autonomic Computing – Degrees, Models, and Applications. *ACM Computing Surveys*, 40(3):1–28, 2008.

[HN96] D. Harel, A. Naamad. The Statemate Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[Hor01] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, 2001. Available at: researchweb.watson.ibm.com/autonomic/manifesto/autonomic_computing.pdf

[HP03] G. Halmans, K. Pohl. Communicating the Variability of a Software-Product Family to Customers. *Software and Systems Modeling*, 2(1):15–36, 2003.

[HS06] M. Hinchey, R. Sterritt. Self-Managing Software. *IEEE Computer*, 33(2):107-109, 2006.

[IBM05] Autonomic Computing Toolkit: Developer's Guide. IBM Technical Report SC30-4083-03. Available at: http://download.boulder.ibm.com/ibmdl/pub/software/dw/autonomic/books/fpy3mst.pdf

[ICAC09] ICAC 2009. *Proceedings of the 6th International Conference on Autonomic Computing*, Barcelona, Spain, June 2009. [Online]. Available at: http://portal.acm.org/citation.cfm?id=1555228

[Jac97] M. Jackson. The Meaning of Requirements. *Annals of Software Engineering*, 3(1):5–21, 1997.

[JJM09] M.A. Jeusfeld, M. Jarke, J. Mylopoulos (Eds.). Metamodeing for Method Engineering. MIT Press, 2009.

[JMF08] I.J. Jureta, J. Mylopoulos, and S. Faulkner. Revisiting the Core Ontology and Problem in Requirements Engineering. In Proc. *16th IEEE International Requirements Engineering Conference (RE 2008)*, Barcelona, Spain, 2008, pp. 71–80.

[JOZ03] S. Jarzabek, W.C. Ong, H. Zhang. Handling Variant Requirements in Domain Modeling. *Journal of Systems and Software*, 68(3):171–182, 2003.

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

[KCHN90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study (CMU/SEI-90-TR-21, ADA235785). Technical Report, SEI/CMU, 1990.

[KK97] P. Kueng, P. Kawalek. Goal-Based Business Process Models: Creation and Evaluation. *Business Process Management Journal*, 3(1):17–38, 1997.

[KKLL99] K.C. Kang, S. Kim, J. Lee, K. Lee. Feature-Oriented Engineering of PBX Software for Adaptability and Reusability. *Software – Practice & Experience*, 29(10):875–896, 1999.

[KL99] V. Kavakli, P. Loucopoulos. Goal-Driven Business Process Analysis Application in Electricity Deregulation. *Information Systems*, 24(3):187–207, 1999.

[KLB05] D. Karastoyanova, F. Leymann, A. Buchmann. An approach to Parameterizing Web Service Flows. In Proc. *International Conference on Service-Oriented Computing (ICSOC 2005),* Amsterdam, Netherlands, December 2005.

[KLMM97] G. Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin. Aspect-Oriented Programming. In Proc. *11th European Conference on Object-Oriented Programming*, 1997, pp. 220–242.

[KLSP01] G. Karsai, A. Ledeczi, J. Sztipanovits, G. Peceli, G. Simon, T. Kovacshazy. An Approach to Self-Adaptive Software based on Supervisory Control. In Proc. *International Workshop on Self-Adaptive Software*, 2001, pp. 24–38.

[KM90] J. Kramer, J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[KM09] J. Kramer, J. Magee. A Rigorous Architectural Approach to Adaptive Software Engineering. *Journal of Computer Science and Technology*, 24(2):183–188, 2009.

[KNS92] G. Keller, M. Nuttgens, A.W. Scheer. Semantische Prozessmodellierung auf der Grundlage “Ereignisgesteuerter Prozessketten (EPK)”. Technical Report 89, Institut fur Wirtschaftsinformatik Saarbrucken, Saarbrucken, Germany, 1992. (In German).

[Knu84] D. Knuth. Literate programming. *The Computer Journal*. 27(2):97–111, 1984.

[KPR04] R. Kazhamiakin, M. Pistore, M. Roveri. A Framework for Integrating Business Processes and Business Requirements. In Proc. *EDOC 2004*, Monterey, USA, 2004.

[Kru95] P. Kruntchen. Architectural Blueprints – the ”4+1” View Model of Software Architecture. *IEEE Software*, 12(6):42–50, 1995.

[KW04] J. Kephart, W. Walsh. An Artificial Intelligence Perspective on Autonomic Computing Policies. In Proc. *IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004, pp. 3–13.

[Lad97] R. Laddaga. Self-adaptive Software. Technical Report 98-12, DARPA Broad Agency Announcement.

[Lam96] A. van Lamsweerde. Divergent Views in Goal-Driven Requirements Engineering. In Proc. *Workshop on Viewpoints in Software Development*, San Francisco, USA, October 1996.

[Lam00] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In Proc. *International Conference on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000.

[Lam01] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In Proc. *5th IEEE International Symposium on Requirements Engineering (RE'01)*, Toronto, Canada, August 2001.

[Lam03a] A. van Lamsweerde. From System Goals to Software Architecture. M. Bernardo, P. Inverardi (Eds.) *SFM 2003*. LNCS Vol. 2804, pp. 25–43, Springer-Verlag, Berlin Heidelberg, 2003.

[Lam03b] A. van Lamsweerde. Goal-Oriented Requirements Engineering: From System Objectives to UML Models to Precise Software Specifications. In Proc. *International Conference on Software Engineering (ICSE 2003)*, Portland, OR, USA, May 3-10, 2003, pp. 744–745.

[Lam04] A. van Lamsweerde. Elaborating Security Requirements by Construction of Intentional Anti-Models. In Proc. *26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, UK, May 2004.

[Lap05] A. Lapouchnian. Goal-oriented requirements engineering: An overview of the current research. University of Toronto, Depth Report, 2005. Available at: http://www.cs.toronto.edu/˜alexei/pub/Lapouchnian-Depth.pdf

[Len98] D. Lenat. The Dimensions of Context-Space. Technical Report, CYC Corp., 1998. Available at: www.cyc.com/doc/context-space.pdf

[LL02] E. Letier, A. van Lamsweerde. Deriving Operational Software Specifications from System Goals. In Proc. *10th Symposium on the Foundations of Software Engineering*, Charleston, USA, November 2002.

[LL03] A. van Lamsweerde, E. Letier. From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering. In Proc. *Radical Innovations of Software and Systems Engineering*, 2003.

[LL04] E. Letier, A. van Lamsweerde. Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering. In Proc. *12th ACM International Symposium on the Foundations of Software Engineering (FSE'04)*, Newport Beach, USA, November 2004.

[LL06] A. Lapouchnian and Y. Lespérance. Modeling Mental States in Agent-Oriented Requirements Engineering. In Proc. *18th Conference on Advanced Information Systems Engineering (CAiSE'06)*, Luxembourg, June 5-9, 2006. E. Dubois and K. Pohl (Eds.), LNCS Vol. 4001, pp. 480–494, Springer-Verlag, Berlin Heidelberg, 2006.

[LL10] A. Lapouchnian and Y. Lespérance. From Adaptive Systems Design to Autonomous Agent Design. In Proc. *4th International* i* *Workshop (istar 2010)*, Hammamet, Tunisia, June 7-8, 2010.

[LLMY05] A. Lapouchnian, S. Liaskos, J. Mylopoulos, Y. Yu. Towards Requirements-Driven Autonomic Systems Design. In Proc. *ICSE 2005 Workshop on Design and Evolution of Autonomic Application Software (DEAS 2005)*, St. Louis, Missouri, USA, May 21, 2005. *ACM SIGSOFT Software Engineering Notes*, 30(4), July 2005.

[LLWY05] S. Liaskos, A. Lapouchnian, Y. Wang, Y. Yu, S. Easterbrook. Configuring Common Personal Software: a Requirements-Driven Approach. In Proc. *13th IEEE International Requirements Engineering Conference*, Paris, France, Aug 29-Sep 2, 2005.

[LLYY06] S. Liaskos, A. Lapouchnian, Y. Yu, E. Yu, J. Mylopoulos. On Goal-based Variability Acquisition and Analysis. In Proc. *14th IEEE International Requirements Engineering Conference*, Minneapolis, USA, Sep 11-15, 2006.

[LM04] D. Lau, J. Mylopoulos. Designing Web Services with Tropos. Proc. *International Conference on Web Services (ICWS'04)*, San Diego, CA, USA, 2004.

[LM09] A. Lapouchnian and J. Mylopoulos. Modeling Domain Variability in Requirements Engineering with Contexts. In Proc. *28th International Conference on Conceptual Modeling (ER 2009)*, Gramado, Brazil, Nov 9-12, 2009. A.H.F. Laender, S. Castano, U. Dayal, F. Casati, J.P.M. de Oliveira (Eds.), LNCS Vol. 5829, pp. 115–130, Springer-Verlag, Berlin Heidelberg, 2009.

[LPH04] H. Liu, M. Parashar, S. Hariri. A Component-based Programming Model for Autonomic Applications. In Proc. *IEEE International Conference on Autonomic Computing*, New York, USA, May 2004, pp. 10–17.

[LRS03] R. Laddaga, P. Robertson, H. Shrobe. Introduction to Self-Adaptive Software: Applications. In Proc. *2nd International Workshop on Self-Adaptive Software*, LNCS Vol. 2614, pp. 275–283, Springer-Verlag, Berlin Heidelberg, 2003.

[LS99] E. Lupu, M. Sloman. Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999.

[LW98] A. van Lamsweerde, L. Willemet. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, December 1998.

[LWZ05] M. Litoiu, M. Woodside, T. Zheng. Hierarchical Model-Based Autonomic Control of Software Systems. In Proc. *ICSE 2005 Workshop on Design and Evolution of Autonomic Application Software (DEAS 2005)*, St. Louis, Missouri, USA, May 21, 2005. *ACM SIGSOFT Software Engineering Notes*, 30(4), July 2005

[LY01] L. Liu, E. Yu. From Requirements to Architectural Design – Using Goals and Scenarios. In Proc. *Workshop From Software Requirements to Architectures (STRAW'2001)*, Toronto, Canada, May 2001.

[LYLM06] A. Lapouchnian, Y. Yu, S. Liaskos, J. Mylopoulos. Requirements-Driven Design of Autonomic Application Software. In Proc. *16th Annual International Conference on Computer Science and Software Engineering CASCON 2006*, Toronto, Canada, Oct 16–19, 2006.

[LYM07] A. Lapouchnian, Y. Yu, J. Mylopoulos. Requirements-Driven Design and Configuration Management of Business Processes. In Proc. *5th International Conference on Business Process Management (BPM 2007)*, Brisbane, Australia, Sep 24-28, 2007. G. Alonso, P. Dadam, and M. Rosemann (Eds.), LNCS Vol. 4714, pp. 246–261, Springer-Verlag, Berlin Heidelberg, 2007.

[MB97] J. McCarthy and S. Buvac. Formalizing Context (Expanded Notes). *Computing Natural Language*. A. Aliseda et al. (Eds.), CSLI Publications, Stanford, CA, 1997, pp. 13–50.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis. Telos: Representing Knowledge about Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[MCN92] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using non-functional requirements: a process-oriented approach, *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.

[MG05] A. Mukhija, M. Glinz. Runtime Adaptation of Application through Dynamic Recomposition of Components. In Proc. *18th International Conference on Architecture of Computing Systems*, 2005, pp. 124–138.

[MK96] J. Magee, J. Kramer. Dynamic Structure in Software Architectures. In Proc. *4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, San Francisco, USA, October 1996, pp. 3–14.

[MM95] J. Mylopoulos, R. Motschnig-Pitrik. Partitioning Information Bases with Contexts. In Proc. *3rd International Conference on Cooperative Information Systems (CoopIS-95)*, Vienna, Austria, May 9-12, 1995.

[MPP08] M. Morandini, L. Penserini, and A. Perini. Towards Goal-Oriented Development of Self-Adaptive Systems. In Proc. *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008)*. ACM Press, Leipzig, Germany, 2008, pp. 9–16.

[MSKC04] P. McKinley, M. Sadjadi, E. Kasten, B.H.C. Cheng. Composing Adaptive Software. *IEEE Computer*, 37(7):56–64, 2004.

[MT97] N. Medvidovic, R.N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. *SIGSOFT Software Engineering Notes*, 22(6):60–76, 1997.

[Mur04] R. Murch. Autonomic Computing. Prentice Hall, 2004.

[NE00] B. Nuseibeh, S. Easterbrook. Requirements Engineering: A Roadmap. In Proc. *Conference on the Future of Software Engineering*, Limerick, Ireland, 2000, pp. 35–46.

[Nil71] N. Nilsson. Problem Solving Methods in Artificial Intelligence. McGraw Hill, 1971.

[OCL06] Object Constraint Language, OMG Specification, Version 2.0, 2006. Available at: http://www.omg.org/cgi-bin/doc?formal/2006-05-01

[OGTH99] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, A. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[OLKM00] R.C. van Ommering, F. van der Linden, J. Kramer, J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2000.

[OOME07] OpenOME. Available at: www.cs.toronto.edu/km/openome/, 2007.

[Par72] D. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *CACM*, 15(12): 1253–1258, 1972.

[Pav00] D. Pavlovic. Towards Semantics of Self-Adaptive Software. In Proc. *International Workshop on Self-Adaptive Software*, LNCS Vol. 1936, pp. 65–74, Springer-Verlag, Berlin Heidelberg, 2000.

[PD90] R. Prieto-Diaz. Domain Analysis: an Introduction. *SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

[PM95] D. Parnas, J. Madey. Functional Documents for Computer Systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[Pot95] C. Potts. Using Schematic Scenarios to Understand User Needs. In Proc. *Designing Interactive Systems (DIS'95)*, Ann Arbor, USA, August 1995.

[PW92] D. Perry, A. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[QP09] N. A. Qureshi and A. Perini. Engineering Adaptive Requirements. In Proc. *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009)*, 2009, pp. 126–131.

[RBPE91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. Object-Oriented Modeling and Design. Prentice Hall, 1991.

[RFP02] S. Robak, B. Franczyk, K. Politowicz. Extending the UML for Modelling Variability for System Families. *International Journal of Applied Mathematics and Computer Science*, 12(2):285–298, 2002.

[RL05] P. Robertson, R. Laddaga. Model-Based Diagnosis and Contexts in Self Adaptive Software. In Proc. *Self-* Properties in Complex Information Systems*, LNCS Vol. 3460, pp. 112–127, Springer-Verlag, Berlin Heidelberg, 2005.

[RN05] S. Russell, P. Norvig. Artificial Intelligent: A Modern Approach. Prentice Hall, 1995.

[Rob89] W. Robinson. Integrating Multiple Specifications Using Domain Goals. In Proc. *5th International Workshop on Software Specification and Design (IWSSD-5)*, Pittsburgh, USA, May 1989.

[Ros77] D. Ross. Structured Analysis: A Language for Communicating Ideas. *IEEE Transactions on Software Engineering*, 3(1):16–34, January 1977.

[Ros06] D. Rosenthal. Consciousness and Mind. Oxford University Press, 2006.

[Roy70] W. Royce. Managing the Development of Large Software Systems. In Proc. 9th ACM/IEEE International Conference on Software Engineering (ICSE 1970), 1970, pp. 328–338.

[RS77] D. Ross, K. Schoman. Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, 3(1):6–15, January 1977.

[RSB98] C. Rolland, C. Souveyet, C. Ben Acour. Guiding Goal Modeling Using Scenarios. *IEEE Transactions on Software Engineering*, 24(12):1055–1071, December 1998.

[SDBF93] W. Spears, K. De Jong, T. Baeck, D. Fogel, H. Garis. An Overview of Evolutionary Computing. In Proc. *European Conference on Machine Learning (ECML-93)*, Vienna, Austria, April 5-7, 1993.

[SEAM09] B.H.C. Cheng, et al. Seams 2009: Software engineering for adaptive and self-managing systems. In Proc. *31st International Conference on Software Engineering (ICSE 2009): Companion Volume*. IEEE Computer Society, Washington, DC, USA, 2009, pp. 463–464.

[Sel03] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*. 20(5):19–25, 2003.

[SESA09] B.H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee (Eds.) *Software Engineering for Self-Adaptive Systems*, LNCS Vol. 5525, Springer-Verlag, Berlin Heidelberg, 2009.

[SG02] B. Schmerl, D. Garlan. Exploiting Architectural Design Knowledge to Support Self-Repairing Systems. In Proc. *14th International Conference on Software Engineering and Data Engineering (SEKE)*, Ischia, Italy, July 15-19, 2002, pp. 241–248.

[SGB05] M. Svahnberg, J. van Gurp, J. Bosch. A Taxonomy of Variability Realization Techniques. *Software – Practice & Experience*, 35(8):705–754, 2005.

[SGM04] R. Sebastiani, P. Giorgini, J. Mylopoulos. Simple and Minimum-Cost Satisfiability for Goal Models. In Proc. *16th International Conference on Advanced Information Systems Engineering (CAiSE 2004)*, Riga, Latvia, LNCS Vol. 3084, pp. 675–693, Springer-Verlag, Berlin Heidelberg, 2004.

[Sim81] H. Simon. The Sciences of the Artificial, 2nd Ed. MIT Press, 1981.

[SKWL99] K. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic Service Matchmaking among Agents in open Information Environments, *ACM SIGMOD Record*, Special Issue on Semantic Interoperability in Global Information Systems, A. Ouksel, A. Sheth (Eds.), 28(1):47–53, 1999.

[SMMM98] A. Sutcliffe, N. Maiden, S. Minocha, D. Manuel. Supporting Scenario-Based Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(12):1072–1088, 1998.

[SM09] V. E. S. Souza and J. Mylopoulos. Monitoring and Diagnosing Malicious Attacks with Autonomic Software. In Proc. *28th International Conference on Conceptual Modeling (ER 2009)*, Gramado, Brazil, Nov 9-12, 2009. A.H.F. Laender, S. Castano, U. Dayal, F. Casati, J.P.M. de Oliveira (Eds.), LNCS Vol. 5829, pp. 84–98, Springer-Verlag, Berlin Heidelberg, 2009.

[SP06] A. Schnieders, F. Puhlmann. Variability Mechanisms in E-Business Process Families. Proc. *International Conference on Business Information Systems* (*BIS 2006)*, Klagenfurt, Austria, 2006.

[ST09] M. Salehie, L. Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, 2009.

[SYN07] M. Salifu, Y. Yu, B. Nuseibeh. Specifying Monitoring and Switching Problems in Context. In Proc. *15th IEEE International Requirements Engineering Conference (RE 2007)*, New Delhi, India, Oct 15-19, 2007.

[SZTJ09] D. Schmitz, M. Zhang, T. Rose, M. Jarke, A. Polzer, J. Palczynski, S. Kowalewski, and M. Reke. Mapping Requirement Models to Mathematical Models in Control System Development. LNCS Vol. 5562, pp. 253–264, Springer-Verlag, Berlin Heidelberg, 2009.

[Tra95] W. Tracz. DSSA (Domain-Specific Software Architecture): Pedagogical Example. *SIGSOFT Software Engineering Notes*, 20(3):49–62, 1995.

[WCLM00] A. Wise, A. Cass, B. Lerner, E. McCall, L. Osterweil, S. Sutton Jr. Using Little-JIL to Coordinate Agents in Software Engineering. In Proc. *15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, Sep 11-15, 2000.

[WMYM09] Y. Wang, S. McIllraith, Y. Yu, J. Mylopoulos. Monitoring and Diagnosing Requirements. *Journal of Automated Software Engineering*, 16(1): 3–35, 2009.

[WSBC09] J. Whittle, P. Sawyer, N. Bencomo, B.H.C. Cheng, and J.-M. Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In Proc. *17th IEEE International Requirements Engineering Conference (RE 2009)*, Atlanta, GA, USA, 2009, pp. 79–88.

[WTKD04] W. Walsh, G. Tesauro, J. Kephart, R. Das. Utility Functions in Autonomic Systems. In Proc. *1st International Conference on Autonomic Computing (ICAC 2004)*, 2004, pp. 70–77.

[YLLM05] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos. Requirements-Driven Configuration of Software Systems. In Proc. *WCRE 2005 Workshop on Reverse Engineering to Requirements (RETR'05)*, Pittsburgh, PA, USA, November 7, 2005.

[YLLM08a] Y. Yu, A. Lapouchnian, J.C.S.P. Leite, J. Mylopoulos. Configuring Features with Stakeholder Goals. In Proc. *23rd Annual ACM Symposium on Applied Computing (SAC 2008)*, RE Track, Brazil, March 16-20, 2008.

[YLLM08b] Y. Yu, A. Lapouchnian, S. Liaskos J. Mylopoulos, J.C.S.P. Leite. From Goals to High-Variability Software Design. In Proc. *17th International Symposium on Methodologies for Intelligent Systems (ISMIS 2008)*, Toronto, Canada, May 20-23, 2008. *Invited Paper*. A. An, et al. (Eds.), Foundations of Intelligent Systems, ISMIS 2008, LNAI Vol. 4994, pp. 1–16, Springer-Verlag, Berlin Heidelberg, 2008.

[YMLL05] Y. Yu, J. Mylopoulos, A. Lapouchnian, S. Liaskos, and J.C.S.P. Leite. From Stakeholder Goals to High-variability Software Designs. Technical Report CSRG-509, University of Toronto, 2005. Available at: ftp://ftp.cs.toronto.edu/csrg-technical-reports/509/.

[Yu93] E. Yu. Modeling Organizations for Information Systems Requirements Engineering. In Proc. *1st IEEE International Symposium on Requirements Engineering*, San Diego, CA, USA, 1993, pp. 34–41.

[Yu97] E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In Proc. *3rd IEEE International Symposium on Requirements Engineering (RE'97)*, January 5-8, 1997, Annapolis, MD, USA. IEEE Computer Society, pp. 226–235.

[Yue87] K. Yue. What Does It Mean to Say that a Specification is Complete? In Proc. *Fourth International Workshop on Software Specification and Design (IWSSD-4)*, Monterey, USA, 1987.

[YWML05] Y. Yu, Y.Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, J.C.S.P. Leite. Refactoring Source Code into Goal Models. In Proc. *13th IEEE International Requirements Engineering Conference (RE'05)*, Paris, France, Aug 29 - Sep 2, 2005.

[XMI07] XML Metadata Interchange (XMI), OMG Specification, Version 2.1.1, 2007. Available at: /www.omg.org/spec/XMI/2.1.1/

[XPAT07] World Wide Web Consortium. XML Path Language (XPath) 2.0 Recommendation, 2007. Available at: www.w3.org/TR/2007/REC-xpath20-20070123/

[XSLT07] World Wide Web Consortium. XSL Transformations (XSLT) 2.0, 2007. Available at: www.w3.org/TR/xslt20/

[Zav97] P. Zave. Classification of Research Efforts in Requirements Engineering. *ACM Computing Surveys*, 29(4):315–321, 1997.

[ZF06] J. Zhang, R. Figueiredo. Autonomic Feature Selection for Appropriate Classification. In Proc. *3rd International Conference on Autonomic Computing (ICAC 2006)*, Dublin, Ireland, 2006.

[ZHJ02] T. Ziadi, L. Helouet, J.-M. Jezequel. Modeling Behaviours in Product Lines. In Proc. *International Workshop on Requirements Engineering for Product Lines (REPL)*, Essen, Germany, September 2002, pp. 33–3.