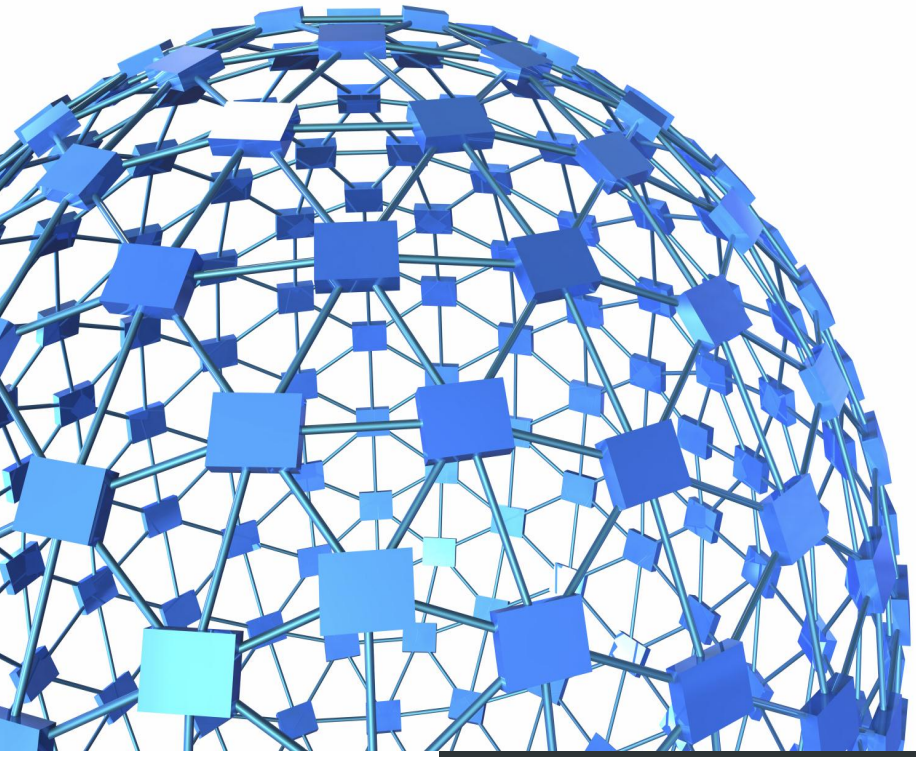
Designing Self-Adaptive Systems through Models at Run-time

María Gómez Lacruz



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Supervisors:
Dr. Joan Fons i Cors
Dr. Vicente Pelechano Ferragud

Centro de Investigación en Métodos de Producción de Software

María Gómez Lacruz

Designing Self-Adaptive Systems through Models at Run-time

Master Thesis, July 2011

PROS
Centro de Investigación en Métodos de Producción de Software

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

María Gómez Lacruz

Designing Self-Adaptive Systems through Models at Run-time

Master Thesis, July 2011

Designing Self-Adaptive Systems through Models at Run-time:

This report was prepared by
María Gómez Lacruz

Supervisors
Dr. Joan Fons i Cors
Dr. Vicente Pelechano Ferragud

Members of the Master Thesis Committee
Dr. Oscar Pastor López, Universidad Politècnica de València
Dr. Vicente Pelechano Ferragud, Universitat Politècnica de València
Dra. Alicia Villanueva García, Universitat Politècnica de València

Centro de Investigación en Métodos de Producción de Software
Universitat Politècnica de València
Camí de Vera s/n, 46022 València
Spain

Tel: (+34) 963 877 007 (Ext. 83530)
Fax: (+34) 963 877 359
Web: http://www.pros.upv.es

Date: July-2011
Comments: A document presented in partial fulfilment of the requirements for the degree of Master in Software Engineering, Formal Methods and Information Systems by Universitat Politècnica de València.

A mis padres y mi hermano

Abstract

Adaptability is emerging as an underlying capability of software systems. There is an increasing demand for systems that dynamically adapt their behavior at run-time in response to changes in the surrounding physical environment, without or with minimal human intervention. These changes that occur while the system is in operation require the system adaptation to occur at run-time. Therefore, new challenges about how to specify, design, verify and realize these systems arise.

Previous research has applied different engineering techniques to build dynamically adaptive systems. These approaches have obtained successful frameworks to support dynamic reconfiguration at run-time. However they lack dedicated design techniques or methodologies to systematically develop self-adaptive systems. The implications and repercussions of run-time reconfigurations are very difficult to foresee and control at design time. Thus, this master thesis provides a model-based method for the systematic design of trustworthy self-adaptive systems. The purpose of the method is (1) to analyze run-time reconfiguration effects at design time, and (2) to automatically refine reconfigurations to avoid faulty situations through execution.

Potential design issues and guidelines to assist engineers in the design of self-adaptive systems have been highlighted. We have identified a

series of design properties (Safe Reconfigurations and Reachability, Redundancy, Reversibility, Contextual Consistency and Contextual Determinism) that capture assertions about desirable behavior of the system. For each property, we have defined refinements to automatically incorporate interesting behavior issues in self-adaptive system specifications. We believe that dealing with these properties is essential for reliable systems as a next step in obtaining self-adaptive systems that fulfill many of the user's needs.

We provide a set of tools that support the design method proposed. These tools ease the modeling of self-adaptive systems, and automate the analysis and refinement of reconfigurations.

To evaluate the approach, we have applied the design method in a Smart Hotel self-adaptive system. This case study has demonstrated the difficulty to foresee the behavior of self-adaptive systems at design time. The proposed approach significantly enhances the ability to develop and maintain self-adaptive systems, by providing mechanisms to analyze run-time reconfiguration effects and automatically refine specifications before execution.

Index

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are becoming an inherent part of daily human life. The increasing complexity, distribution, and dynamism of current software systems, has led the software engineering community to look for new ways to design and manage systems and services. In this attempt, the capability of the system to adjust its behavior in response to the environment in the form of **self-adaptation** has become one of the most promising research directions (Brun et al., 2009). The *"self-"* prefix indicates that the system is able to perform changes autonomously, without or with minimal human intervention, during its execution.

There is an increasing demand for software **systems that dynamically adapt their behavior at run-time** in response to changes in their user preferences, requirements, supporting computing infrastructure and in the surrounding physical environment (Cheng et al., 2009; Salehie & Tahvildari, 2009). Changes can also be induced by failures or unavailability of parts of a software system itself. In these circumstances, it is necessary for a software system to change its structure and/or behavior as necessary to continue achieving its goals. These

changes that occur while the system is in operation require the system adaptation to occur at run-time. This poses challenging question about how to specify, design, verify and realize this system (Zhang & Cheng, 2006).

This thesis provides an approach to **design self-adaptive systems**. The approach gives a way for modeling how a system should self-adapt to avoid faulty situations, and guaranteing a series of desirable properties through execution. In this work, modeling techniques are applied in order to face the development of such complex systems from a higher abstraction level.

The reminder of this chapter is organized as follows: Section 1.1 explains the purpose of this work. Section 1.2 details the problem that the present thesis resolves. Section 1.3 introduces the goals defined for this work. Section 1.4 describes the approach followed in this thesis to fulfill the detected goals. Section 1.5 introduces the research methodology that has been followed. Section 1.6 explains the context in which the work has been performed. Finally, Section 1.7 gives an overview of the structure of this document.

1.1 Motivation

Self-adaptability is emerging as a necessary underlying capability for many applications, particularly for areas such as environmental monitoring, disaster management and other applications deployed in dynamically changing environments (Bencomo et al., 2008a). Such applications inevitably have to reconfigure themselves according to fluctuations in their environment. Current software development approaches are not adequate to develop systems that dynamically adapt to environment fluctuations. As a result, innovative alternatives are required (Bencomo et al., 2008b).

The engineering of self-adaptivity is one of the most challenging issues to address in current systems due to its complexity, as well as the unpredictability of changes in the environment. Although numerous research efforts (Cetina et al., 2008; Parra et al., 2009; Trinidad

et al., 2007; Hallsteinsen et al., 2006; Istoan et al., 2009) have focused on designing self-adaptive software, there are still some major open problems in this area.

Previous research has applied different engineering techniques to build dynamically adaptive systems. These approaches have obtained successful frameworks to support dynamic reconfiguration at run-time. However they lack dedicated design techniques or methodologies to systematically develop reconfigurable systems from requirements to a validated and verified reconfigurable system.

A fundamental principle of self-adaptive systems is variability management. Current approaches use variability modeling from design-time at run-time. In general terms, the development of self-adaptive systems first focus on characterizing the variability by means of variability models and, afterwards, leverage those models at run-time to allow the system to adapt itself. Nevertheless, dynamic adaptation can often lead to inappropriate or unpredictable behavior. For traditional software systems, once a product is obtained, it can be tested intensively before it reaches the users. However, the case of self-adaptive systems is different since different configurations are obtained at run-time. Therefore, it is difficult to evaluate the design of an adaptive system and also it is difficult to test the system behavior since it depends on specific events or context situations. In addition, a failure in reconfigurations directly impacts the user experience because the reconfiguration is performed when the system is already under the user's control (Cetina et al., 2010).

The goal of this master thesis is to develop an appropriate approach to model, analyze and validate the behavior of self-adaptive systems before its execution.

1.2 Problem Statement

In spite of numerous excellent research efforts, the development of self-adaptive systems is still in its infancy. The work presented in this thesis seeks to improve the development of self-adaptive systems by focusing on the design stage. In particular, the challenges that this

thesis addresses can be stated by the following three research questions:

Research question 1. How to carry out a systematic design approach to develop trustworthy self-adaptive systems?

Research question 2. How to specify assertions about desirable behavior that self-adaptive systems should perform during execution?

Research question 3. How can dynamic adaptation be validated and adjusted at design-time to avoid failures during execution?

These research questions are analyzed and answered in the following sections.

1.3 Thesis Goals

The main goal of this master thesis is to define a methodological approach to systematically design self-adaptive systems.

First of all, **research question 1** deals with the important issue of design for adaptability. This work defines a systematic design method to build complex and dependable self-adaptive software systems. Techniques and tools are provided in order to aim software engineers to minimize error-proneness of this kind of complex development.

Regarding **research question 2**, given the adaptation specification of self-adaptive systems it is difficult to foresee the behavior that the system will perform at run-time. We have identified a series of design properties that capture assertions about desirable behavior of the system. For each property, we have defined refinements to automatically incorporate interesting behavior issues in adaptation specifications. We believe that dealing with these properties is essential for reliable systems as a next step in obtaining self-adaptive systems that fulfill many of the user's needs.

Regarding **research question 3**, another goal of this work is to determine how to validate the behavior that the system will perform

at run-time. Since the models that form the basis for reconfiguration strategies are available at design time, it is possible to conduct a thorough analysis to validate configurations in an early stage of the development process without first implementing them. This work provides support for validating adaptation, if problems are foreseen, the models can be adjusted through refinements to avoid the appearance of failures during execution. We have automated this step with tool support.

1.4 The Proposed Solution

Model Driven Engineering (MDE) (Schmidt, 2006) proposes the use of models as the basis for system development. A model is a simplification of a system, built with an intended goal in mind, that should be able to answer questions in place of the actual system (Bézivin & Gerbé, 2001). The use of models in engineering has a twofold benefit. On the one hand, models guide the development of a system. On the other hand, models allow to reason about the system avoiding to deal with technical details.

In a context where the possible combinations of the system and the context situations are constantly increasing, the implementation of ad-hoc solutions to cover all possible combinations is not feasible. At design time, engineers can **avoid designing by hand** all of the system's possible configurations and transitions by explicitly defining a self-adaptive system as a Dynamic Software Product Line (DSPL)(Morin et al., 2009). Engineers model the dynamic adaptation of the system and the possible variants as a DSPL. These models are defined offline before the initial system deployment, and are leverage at run-time to drive the dynamic adaptation process. The quality and correctness of models are crucial and must be checked as early as possible.

In this work, we introduce a design process based on the foundations of MDE and DSPL to design dependable self-adaptive systems. Specifically, this approach provides the following contributions:

A **design method** is defined to guide engineers in the construction of trustworthy self-adaptive systems in a systematic way. The method

comprises from specification to the final running system. The purpose of the design method is (1) to analyze the run-time reconfiguration effects with respect to a set of stated valuable properties (2) to assure that the system perform the desirable behavior at run-time through refinements. We have proposed several design properties to manage important concerns of system execution. Dealing with design properties optimizes self-adaptive system design and, consequently, execution since the design models are also used at run-time to drive the behavior of the system.

1.5 Research Methodology

In order to perform the work of this thesis, we have followed the design methodology for performing research in information systems as described by (March & Smith, 1995) and (Vaishnavi & Kuechler, 2004). Design research involves the analysis of the use and performance of designed artifacts to understand, explain and, very frequently, to improve on the behavior of aspects of Information Systems (Vaishnavi & Kuechler, 2004).

The design cycle consists of 5 process steps: (1) awareness of the problem, (2) suggestion, (3) development, (4) evaluation, and (5) conclusion. The design cycle is an iterative process; knowledge produced in the process by constructing and evaluating new artifacts is used as input for a better awareness of the problem.

Following the cycle defined in the design research methodology, we started with the awareness of the problem (see Figure 1.1): we identified the problem to be resolved and we stated it clearly.

Next, we performed the second step which is comprised of the suggestion of a solution to the problem, and comparing the improvements that this solution introduces with already existing solutions. To do this, the most relevant approaches were studied in detail. Once the solution to the problem was described, we plan to develop and validate it (steps 3 and 4). These two steps will perform in several phases (see Figure 1.1).

Finally, we will analyze the results of our research work in order

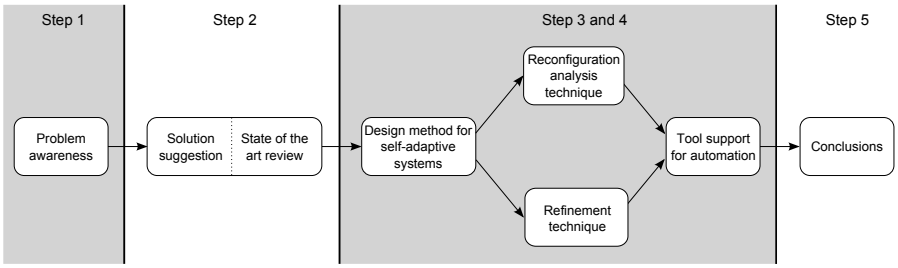


Figure 1.1: Research methodology followed in this thesis

to obtain several conclusions as well as to delimitate areas for further research (step 5).

1.6 Thesis Context

This Master Thesis has been developed in the context of the research center *Centro de Investigación en Métodos de Producción de Software (ProS)* of the *Universitat Politècnica de València*. The work that has made the development of this thesis possible is in the context of the following research projects:

- EVERYWARE: project referenced as TIN2010-18011, supported by Ministerio de Ciencia e Innovación (MICINN).
- VALi+d: program for training of researchers, supported by Conselleria d'Educación of Generalitat Valenciana.

1.7 Thesis Structure

Figure 1.2 shows a roadmap for this thesis. It consists of eight chapters as follows:

Chapter 2, *Background*. This Chapter presents the main concepts and characteristics of the approaches related with this thesis, in order to

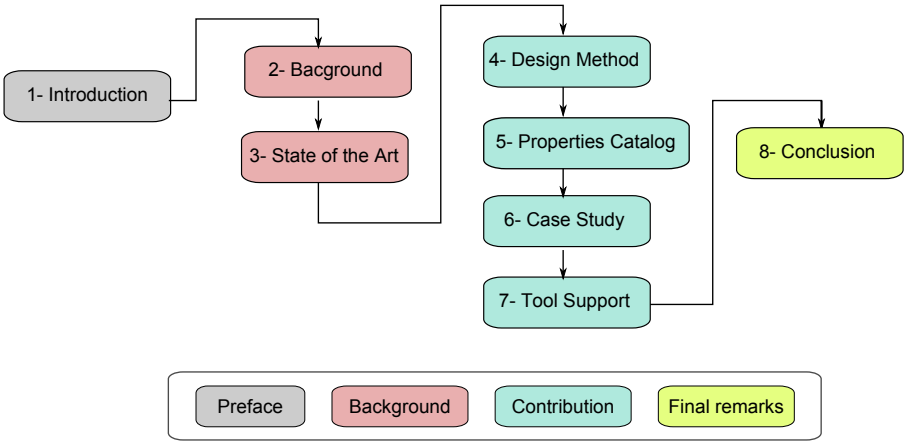


Figure 1.2: Roadmap of this master thesis

provide to the reader a basic background for understanding the overall thesis work. Specifically, this chapter presents Autonomic Computing, Model Driven Development and Software Product Lines.

Chapter 3, *State of the Art*. This chapter shows an analysis of the most important approaches that have been proposed to face self-adaptivity.

Chapter 4, *A design method for Self-Adaptive Systems*. This chapter introduces the design method for the development of self-adaptive systems through variability refinementsw. This overview covers the main building blocks of the approach as well as the process to apply it.

Chapter 5, *Properties Catalog*. This chapter terms a catalog of desirable design properties for self-adaptive systems and particular refinements to assure each property automatically.

Chapter 6, *Case Study*. This chapter introduces how the approach has been evaluated throughout the case study of a Smart Hotel.

Chapter 7, *Tool support*. This chapter shows an overview of the tools proposed to support the design method.

Chapter 8, *Conclusions and Future Work*. This chapter presents

the main contributions and results of this work. In addition, this chapter discusses future research directions.

Chapter 2

Background

In this chapter the background of the master thesis is introduced. This thesis deals with the design of self-adaptive systems that can be adapted when context conditions change. As it is shown in Figure 2.1, our approach combines aspects of three research areas: **Autonomic Computing**, **Models@run-time** and **Dynamic Software Product Lines**. Therefore, this chapter presents the main concepts and characteristics of these disciplines in order to provide a basic background for understanding the overall thesis work.

The rest of this chapter is organized as follows. Section 2.1 presents the foundations of *Autonomic Computing*. Section 2.2 presents *Model Driven Development*, which is a paradigm where we can construct a model of a software system that can then be transformed into the real functional software. In addition, the *Models@run-time* approach is introduced as a subfield of the previous one. Next, 2.3 introduces *Software Product Lines* as a paradigm of software reuse which intends to produce a set of products that share a common set of assets in an specific domain. Furthermore, *Dynamic Software Product Lines*, are detailed as a

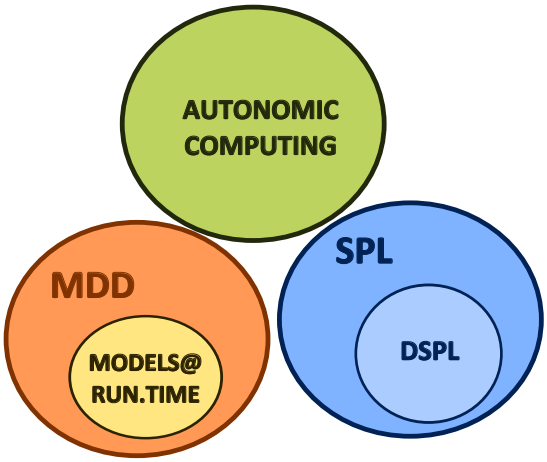


Figure 2.1: Approaches involved in this master thesis

special case of Software Product Lines. Finally, Section 2.4 concludes the chapter.

2.1 Autonomic Computing

In 2001, IBM released a manifesto (Horn, 2001) describing the vision of Autonomic Computing. The purpose is to countermeasure the complexity of software systems by making systems self-managing. The paradox has been spotted, that systems need to become even more complex to achieve this. The complexity, it is argued, can be embedded in the system infrastructure, which in turn can be automated. The similarity of the described approach with the autonomic nervous system of the body, which relieves basic control from our consciousness, gave birth to the term Autonomic Computing.

In Kephart's and Chess' Vision of Autonomic Computing (Kephart & Chess, 2003), the following four fundamental properties about autonomic computing were discussed: *self-configuration*, *self-optimization*, *self-healing* and *self-protection*. All of these four fundamental proper-

ties work together to build the essence of autonomic computing: *self-management*. As a result, the system components and devices will seem completely natural and the users are unaware of the complexity of entire autonomic computing system. Now the self-* properties has grown into a wide range list (Sterritt et al., 2005). Here is a brief description of the fundamental properties:

Self-configuration. An autonomic computing system configures itself according to high-level goals, that is, by specifying what is desired, not necessarily how to accomplish it. This can mean being able to install and set itself up based on the needs of the platform and the user.

Self-optimization. An autonomic computing system optimizes its use of resources. It may decide to initiate a change to the system proactively (as opposed to reactive behavior) in an attempt to improve performance or quality of service.

Self-healing. Autonomic computing systems should have the ability to discover, diagnose and repair failed components without the disruption of system services. Systems and critical services must not be crashed, disrupted even under the extreme conditions. To achieve it, autonomic computing systems must have the ability to recovery from an unexpected situation automatically.

Self-protection. Autonomic computing systems should have the ability to anticipate, detect, identify and protect against attacks from anywhere. Autonomic systems will be self-protecting in two senses. They will defend the system as a whole against large-scale, correlated problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures. They also will anticipate problems based on early reports from sensors and take steps to avoid or mitigate them.

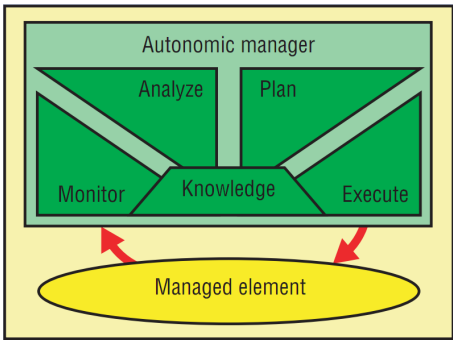


Figure 2.2: IBM's MAPE-K reference model for autonomic control loops (Kephart & Chess, 2003)

2.1.1 The MAPE-K Autonomic Loop

To achieve autonomic computing, IBM has suggested a reference model for autonomic control loops (Kephart & Chess, 2003), which is known as MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop and is depicted in Figure 2.2.

In the MAPE-K autonomic loop, the managed element represents any software or hardware resource that is given autonomic behavior by coupling it with an autonomic manager. The managed element can for example be a specific part or software component or a complete system. The data collected by the sensors allows the autonomic manager to monitor the managed element and execute changes through effectors. The autonomic manager is a software component that ideally can be configured by human administrators using high-level goals and uses the monitored data from sensors and internal knowledge of the system to plan and execute, based on these high-level goals, the low-level actions that are necessary to achieve these goals.

2.2 Model Driven Development

Model Driven Development(MDD) or Model Driven Engineering (MDE) is a modern software development paradigm where applications are specified at a more abstract level using models. Then, by means of transformation techniques, one or more concrete implementations are generated (Stahl et al., 2006; Bézivin, 2004).

With the increase of complexity of today software systems, MDD has become an accepted and widely applied development approach providing a number of benefits over more traditional, code oriented, software development. Through the utilization of MDD, software developers are able to separate specific domain knowledge from technology concerns. Formal models and transformations allow traceability between models and their target implementations, providing reliable links through system requirements, design, implementation and testing. Thus, MDD maximizes the productivity and quality of software.

Model Driven Architecture (MDA) is a framework proposed by the Object Management Group (OMG) in 2001 to provide standard guidelines in engineering software systems with MDD approaches (Miller & Mukerji, 2003). MDA does not explicitly specify a detailed development process, rather it gives only generic method guidelines defining the system models and facilitating transformations between different model types. MDA specifies system in three layered models: system requirements are specified in the **CIM** (Computation-Independent Model); the Platform-Independent Model (**PIM**) is the model that describes the system design independent of the implementation platform; the Platform-Specific Model (**PSM**), on the other hand, describes the system design in the form of a platform-dependent model. From PIM following an OMG standard mapping, PSMs are generated from where application code is generated. Figure 2.3 illustrates the software development lifecycle according to MDA.

MDA is actually a consolidation of several OMG standards for using models extensively in software development. PIMs are defined using platform independent modeling languages like UML (Unified modeling language). UML Profile which is a standardized set of extensions for

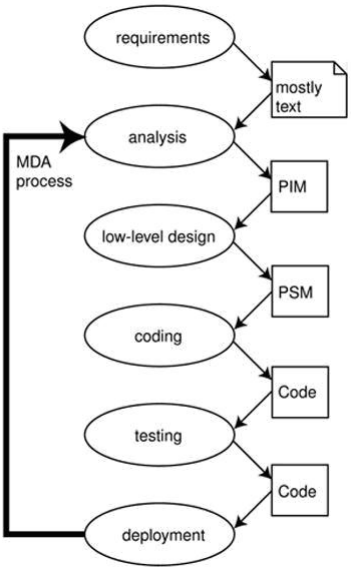


Figure 2.3: MDA software development lifecycle (Miller & Mukerji, 2003)

UML models to define PIMs with a specific domain details. For performing automated mapping between different models a model based transformation language QVT (OMG, 2005) is a standard. Meta Object Facility (OMG, 2006) is a standard for defining metamodels for other languages like UML. These and several other standards define the core infrastructure of the MDA and provide the architecture for modern system modeling.

2.2.1 Models@run-time

In the model-driven software development area, research effort has focused primarily on using models at design, implementation and deployment stages of development. Nevertheless the software models produced during development can be effectively leveraged during run-time (Blair et al., 2009). A promising approach to manage complexity in run-time environments is to develop adaptation mechanisms that leverage soft-

ware models, referred to as models@run.time (Morin et al., 2009).

The aim of models@run.time is to extend the applicability of model-driven engineering (MDE) techniques to the run-time environment. The models@run.time approach considers a model in the same way that defines MDE but with some slight differences. A run-time model is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective.

Models at run-time are considered as a key enabling technology for systems that control themselves as they operate. Run-time models can be used in different ways: to support dynamic state monitoring and control of systems during execution, to support semantic integration of heterogeneous software elements at run-time, to fix design errors, to fold new design decisions into a running system to support controlled ongoing design, to support unanticipated modifications and to support dynamic evolution of software design (Blair et al., 2009).

Summarizing, the use of models at run-time brings new opportunities for autonomic capabilities by reutilizing the efforts invested at design time. The modeling effort made at design time is not only useful for producing the system but also provides a richer semantic base for autonomic behavior during execution.

2.3 Software Product Lines

One increasing trend in software development is the need to develop multiple, similar software products instead of just a single individual product. Because of cost and time constraints it is not possible for software developers to develop a new product from scratch for each new customer, therefore, software reuse must be increased.

The Software Engineering Institute (SEI) proposed Software Product Line Engineering (SPLE) as a solution to increase reuse in software development. A Software Product Line (SPL) is a set of software systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed

from a common set of core assets in a prescribed way (Clements & Northrop, 2001). SPL is rapidly emerging as a viable and important software development paradigm allowing companies to realize improvements in time to market, cost, productivity, quality, and other business drivers.

A fundamental characteristic of product line engineering is a shift from a single system point-of-view to a set of products. As a result, the differences or variations among products become a primary concern. Thus management of variation, so called **variability management**, is a core capacity of SPLE. The idea is to separate all products in the product line into three parts and to manage them throughout development:

Commonality: parts which are common to all products in the product line. On the requirements side these are common requirements; in the implementation this results in common components.

Reusable variation: aspects that are common to some, but not all, products in the product line. These are the powerhouse of product line engineering, by providing a plug-and-play mechanism, it is easy to assemble new products by reuse of existing variability.

Product specifics: no matter how well designed a product line is there will always be requirements that are specific to individual products. Here, it is key not to waste any effort on generic development for aspects that will be used only once.

In addition to variability management, a second key principle of product line engineering is the use of a two-lifecycle approach. Figure 2.4 illustrates the SPL development approach. The development is separated into Domain Engineering process and Application Engineering process:

Domain Engineering is responsible for all analysis of the product line as a whole and for producing any common and reusable variable parts.

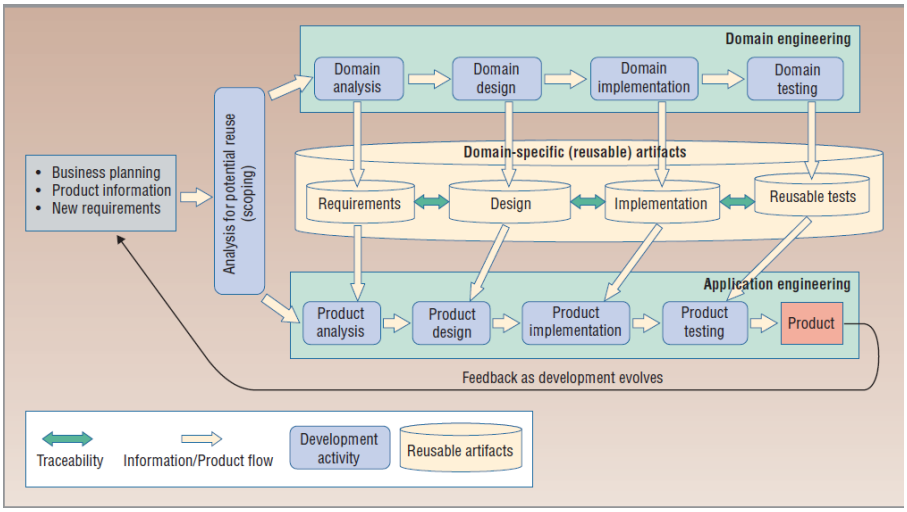


Figure 2.4: SPL life-cycle approach (Hallsteinsen et al., 2008)

Application Engineering is then responsible for the production of product-specific parts, the integration and for all aspects particular to individual products.

The approach has been successfully used to develop a wide variety of product lines in a number of different domains, ranging from avionics over medical equipment to information systems, in a wide variety of organizations.

2.3.1 Dynamic Software Product Lines

In emerging domains such as ubiquitous computing, service robotics or ambient intelligence, applications are increasing in complexity, and a higher degree of adaptability is demanded (Istoan et al., 2009). To cope with these challenges, there is a need for new development techniques that produce software capable of adapting to dynamic environments.

Dynamic Software Product Lines (DSPLs) (Hallsteinsen et al., 2008)

provides a systematic way to build self-managing systems, i.e., software systems that can optimize themselves relative to changes in their operating environment and in customer needs. DSPLs exploits traditional SPL concepts (such as variability modeling, common assets shared by product variants, and automated product derivation) to provide a systematic basis for the engineering of adaptive systems.

The central shift from the traditional view on software product lines to dynamic software product lines is that variation points are bound at run-time, first when software is launched to adapt it to the current situation, and subsequently re-bound during operation to adapt the software to changes in the situation. Although traditional SPL engineering recognizes that variation points are bound at different stages of the development, and possibly also at run-time, the focus has been on variation points that are bound before the software is delivered.

The main idea of DSPLs is to achieve systems capable of modifying their own behavior with respect to changes in their operating environment by dynamically rebinding variation points at run-time. In a DSPL, a configurable product (CP) is produced from a product line similarly to standard SPL. However, the reconfiguration ability implies the usage of two artifacts to control it: the decision maker and the reconfigurator. The **decision maker** is in charge of capturing all the information in its environment that suggests a change such information from external sensors or even from users. The analyzer must know the whole structure of a CP so it makes a decision on which features must be activated and deactivated. The **reconfigurator** is responsible of executing the decision by using the standard SPL run-time binding. A CP may be considered as an extension to traditional SPL products where there are no bound features but the decision maker and the reconfigurator and the remaining features are bound at run-time. As a consequence, new features may be added to an existing product or even existing features may be updated at run-time.

In summary, a DSPL should have the following properties:

- It exhibits dynamic (run-time) variability: (re-)configuration and binding at run-time;

- Variable properties change several times during lifetime;
- It deals with environmental changes;
- It deals with changes from users (e.g., changes in functional and/or quality requirements);
- It may exhibit changes of variation points during run-time, e.g., addition of variation points during run-time (optional);
- It may support context-awareness (optional), situation awareness;
- It behaves autonomic and exhibits self-adaptive properties (optional);
- It is capable of automatic decision making (optional);
- It may be a subset of SPL that has an individual environment/-context situation.

Interest in DSPLs is growing as more developers apply the SPL approach to dynamic systems e.g (Cetina et al., 2008; Parra et al., 2009; Trinidad et al., 2007; Hallsteinsen et al., 2006; Istoan et al., 2009).

2.4 Conclusions

This chapter introduces the foundations in which our approach relies. It provides an overview of different techniques that are related with the work developed in this master thesis. The analysis has considered three disciplines: Autonomic Computing, Model Driven Development and Software Product Lines.

Chapter 3

State of the Art

This chapter introduces the most important approaches that support design and development of self-adaptive systems. Once we have carried out a thorough analysis, we have identified a series of weaknesses in current approximations. The work proposed in this master thesis deals with the identified weaknesses.

The reminder of the Chapter is organized as follows. Section 3.1 presents different approaches proposed to build self-adaptive systems. Section 3.2 summarizes the conclusions extracted after analyzed these approximations. Finally, Section 3.3 concludes the chapter.

3.1 Self-adaptive systems development approaches

In this section we have analyzed the most relevant approaches for development of self-adaptive systems. The approaches are presented chronologically according to the year in which they appeared.

3.1.1 Hallsteinsen et al. Approach

Hallsteinsen et al. (Hallsteinsen et al., 2006) present the MADAM approach to build adaptive systems. The MADAM approach is based on ideas from software product line engineering. Adaptive applications are built as component based system families with the variability modeled explicitly as part of the family architecture. MADAM uses property annotations on components to describe their Quality of Service. At runtime, the adaptation is performed using these properties and a utility function for selecting the component that best fits the current context. The adaptation platform of the MADAM approach provides (1) a conceptual model and (2) reference architecture for adaptive applications as follows.

1. **The conceptual model** (see Figure 3.1 left) is based on entities which interact with other entities by providing and making use of services through ports. A port represents a service offered by an entity or a service needed by an entity. Entities may be composed of smaller entities, allowing for a hierarchic structure. To model variation, both in the application and in its context, the conceptual model provides the concept of entity type. An entity type defines a class of entities with equivalent ports which may replace each other in a system. With these concepts the conceptual model is able to model an adaptive application architecture as a possibly hierarchic composition of entity types, which defines a class of application variants as well as a class of contexts in which they may operate.

2. **The reference architecture** (see Figure 3.1 right) provides components for monitoring user needs and available resources, for deriving a more suitable variant when the user needs or available resources change, and for transforming the current variant into the preferred one by reconfiguration at the component level. To enable the derivation of the variant that best fits a given context, the MADAM approach is based on property annotations associated with ports. Property annotations allow to reason about how

well an application variant matches its context, by comparing the properties of the services provided by the application with the properties required by the user. The match to user needs is expressed in a utility function. By default the utility function is a weighted mean of the differences between properties representing user needs and properties describing the service provided by the application, where the weights represent priorities of the user.

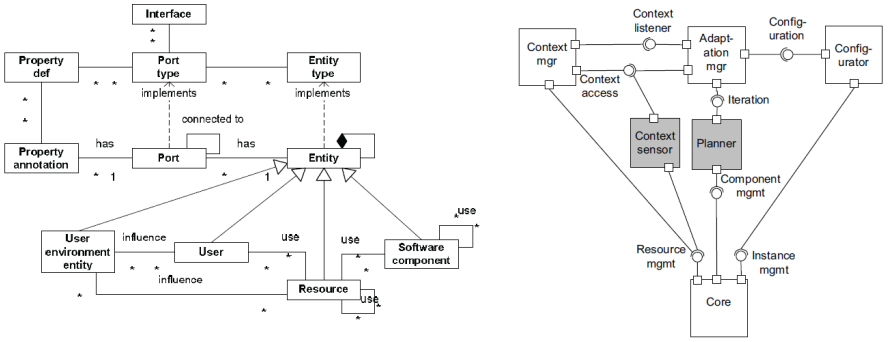


Figure 3.1: Hallsteinsen et al. approach

3.1.2 Morin et al. Approach

Morin et al. (Morin et al., 2008) present a new approach where they address challenges in adaptive system construction and execution by combining certain aspect-oriented and model driven techniques. Models cope with complexity through abstractions and are used both to specify the dynamic variability at design time and to manage run time adaptations. The variant models capture the variability of the adaptive application. The actual configurations of the application are built at run-time by selecting and composing appropriate variants. An adaptation model specifies which variant have to be selected depending on the context of the running application. Specifically, they propose to model the variants instead of the configurations. Then, the configurations can

then be built by automatically combining the variants. In practice this is achieved using Aspect-Oriented Modeling techniques for architecture models. Aspect oriented techniques are utilized to model the adaptation concerns separately from the other aspects of the system.

Figure 3.2 presents the conceptual model of this approach. The approach is divided in two phases; design time and run-time. At design-time, the application base and variant architecture models are designed and the adaptation model is built. At run-time, the adaptation model is processed to produce the system configuration that should be executed.

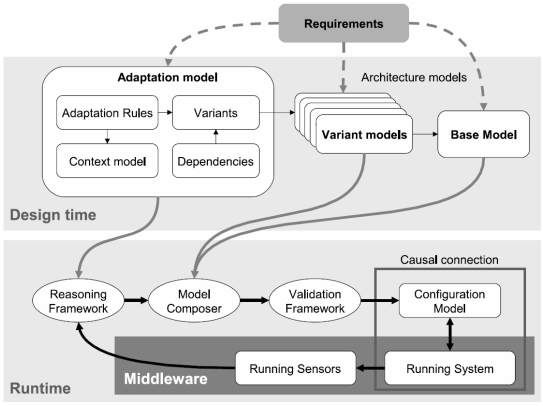


Figure 3.2: Morin et al. approach overview

3.1.3 Parra et al. Approach

Parra et al. (Parra et al., 2009) introduces a Context-Aware Dynamic Software Product Line for building service oriented applications and adapting them at run-time in accordance with their using context. This DSPL, named CAPucine for Context-Aware Service-Oriented Product Line is based on a model-driven approach. CAPucine distinguish two different processes for product derivation. The first process uses a feature model (see Figure 3.3) that represents the product family. For

every feature in the Feature model, there is an associated asset that corresponds to a partial model of the product itself. The assets, represented as models, get composed and transformed in order to generate the product. The second process relates to dynamic adaptation. This process introduces context-aware assets that operate at run-time. These context-aware assets contain three kinds of data: the context when the assets can be modified, the place where the assets must be applied and the change that must be performed. The realization of these context-aware assets combines two run-time platforms. On the one hand, COSMOS is a context-aware framework connected to the environment by the use of sensors. On the other hand FraSCAti is a Service Component Architecture (SCA) platform with dynamic properties that enables to bind and unbind components at run-time. CAPucine allows designing and processing context-aware applications based on an SCA platform which is dynamic, introspectable, and reconfigurable in accordance with the context environment.

Overall, the platform architecture is depicted in Figure 3.3 right. The Context Manager element is composed of several nodes. Every node is in charge of recovering context information from different sources like a sensor layer who captures raw data from the environment, user preferences, and the Run-time Platform who provides information about current state and configuration of applications. Eventually, the Context Manager can also perform a precessing of data, so that, it is presented as single values which can be evaluated in the condition of each context-aware asset. The Decision Maker element is in charge of evaluating the context and decide whether or not to modify the application. It is linked to a repository of rules. The rules represent the clauses of each context-aware asset. Finally, the Run-time Platform element is where Application Components are executed. It controls the life cycle of all the application components and has access to their control mechanisms.

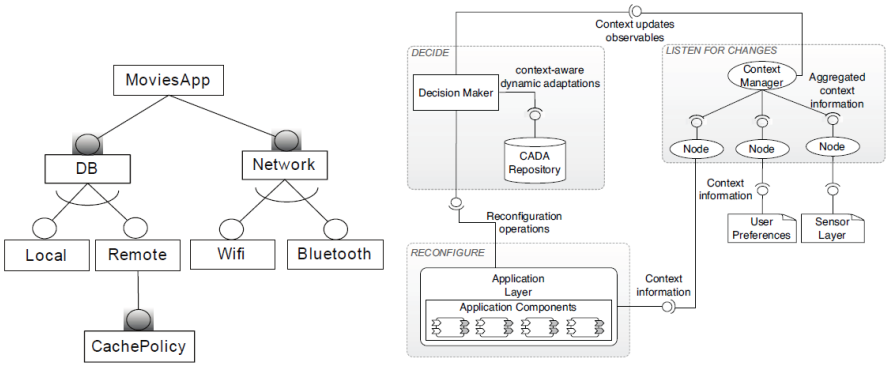


Figure 3.3: Parra et al. approach

3.1.4 Cetina et al. Approach

Cetina et al. (Cetina et al., 2009a) designs self-adaptive pervasive systems by combining the main ideas of Model Driven Development and Dynamic Software Product Lines. This approach reuses design variability models during run-time to achieve autonomic behavior. The run-time variability models support the self-reconfiguration of systems when triggered by changes in the environment. The question when to adapt is solved at run-time when checking a set of context conditions. The approach performs reconfiguration in terms of features and uses a Model-based Reconfiguration Engine (MoRE) to translate contextual changes into changes in the activation/deactivation of features. Then, these changes are translated into the reconfiguration actions that modify the system components accordingly. Cetina et al. have applied their approach to the smart-homes domain.

The overall reconfiguration steps are outlined in Figure 3.4. The Context Monitor uses the run-time state as input to check context conditions (step 1). If any of these conditions are fulfilled (e.g., home becomes empty), then MoRE uses model operations to query the run-time models about the necessary modifications to the architecture (step 2). The response of the models is used by the engine to elaborate a

Reconfiguration Plan (step 3). This plan contains a set of Reconfiguration Actions, which modify the system architecture and maintain the consistency between the models and the architecture (step 4). The execution of this plan modifies the architecture in order to activate/deactivate the features specified in the resolution (step 5).

The reconfiguration of the system is performed by executing reconfiguration actions that deal with the activation/deactivation of components, the creation and destruction of channels among components and the update of models accordingly to keep them in sync with the system state. MoRE makes use of the OSGi framework for implementing the reconfiguration actions. This Framework implements a complete components model that extends the dynamic capabilities of Java.

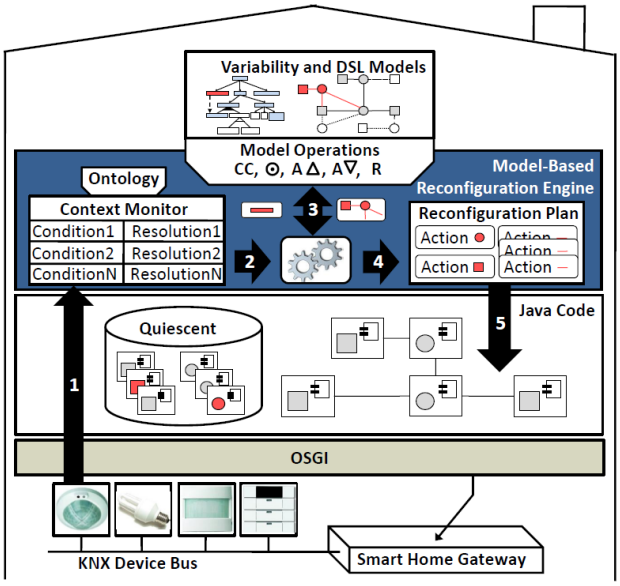


Figure 3.4: Cetina et al. approach

3.2 Analysis and Discussion

The different approaches in the self-adaptiveness area are mainly focused on supporting reconfiguration at run-time. These proposals normally observe events form the environment and provide automatic reconfiguration of the system. According to the approaches evaluated in this chapter, we have identified common weaknesses that this master thesis tries to solve:

- The approaches provide an execution platform or mechanism to realize reconfiguration at run-time, but they lack a dedicated methodology enabling developers to systematically develop the reconfigurable systems. That is, developer guidance from the requirements to a validated and verified reconfigurable system.

- The approaches lack support for notable concerns such as validation analysis, debugging or tracking capabilities.

- All of the analyzed approaches are also missing the evaluation of the safety and reliability of run-time reconfigurations. These properties are essential for the development of reliable self-adaptive systems.

The approach proposed in this master thesis complements the previous approaches by providing a design method that deals with the identified weaknesses.

3.3 Conclusions

This chapter presents the state of the art in the disciplines that are related to this work. Although self-adaptiveness has become increasingly interesting and popular, its achievement in a systematic manner it remains a relatively immature topic. There is still a lack of proposals to provide mechanisms that allow the development of self-adaptive systems from requirements to a validated system.

The present work provides support for the design and validation of self-reconfiguration at run-time. Since many approaches already address them, our approach is not focused in achieve autonomic behavior but describing the ways in which self-adaptive systems must be developed to guarantee valuable issues at run-time. We believe it is indispensable to come to a software engineering approach which supports self-adaptive system engineers from design time to run-time addressing the former open challenges.

Chapter 4

A Design Method for Self-Adaptive Systems

Increasingly, software needs to dynamically adapt its behavior at run-time in response to changing conditions in the supporting computing, communication infrastructure, and in the surrounding physical environment (McKinley et al., 2004). Self-adaptive software systems are able to reconfigure their structure and modify their behavior at run-time in order to adapt to environmental changes, repair faults, or optimize their operation. Development of adaptive systems introduces additional challenges compared to the development of traditional software systems. Adaptive systems are generally more difficult to specify, verify, and validate due to their high complexity. Therefore, new innovative approaches for building, running, and managing dynamically reconfigurable systems are required.

This chapter introduces a methodological approach for the design of self-adaptive systems. The goal of this method is to provide a mechanism to automatically achieve well-designed adaptive systems through systematic specification refinements. We mean as well-designed sys-

tems, the ones that will never produce errors during adaptation and fulfill established assertions about design.

We combine model-driven and DSPLs to better cope with the complexities during the construction and execution of adaptive systems. The use of these techniques allows us to use high-level abstractions and simplify the representation and management of system variants. We use models at run-time (Blair et al., 2009) as if they were the policies that drive the autonomic adaptation of the system at run-time. One of the main advantages is that the adaptation rules do not have to be manually written for each possible reconfiguration. The specification obtained at design drives the later system execution. Therefore, techniques to validate the adaptation of the system at design-time are needed to prevent the occurrence of errors during execution. We provide mechanisms to validate the adaptation at design-time. If during the validation process errors arise, the model is refined to avoid such problems. The model validated (and refined if needed) at design-time is used at run-time to drive the system behavior safely.

The reminder of the chapter is structured in the following manner. Section 4.1 first introduces the concepts used in the method to describe the self-adaptive system in an abstract manner, and then provides an overview of the design method. Section 4.2 provides detail on the stages involved in the design method. Finally, Section 4.3 concludes the chapter.

4.1 Proposal Overview

To deal with the challenges of adaptation, the design method proposed in this master thesis builds on the central ideas of Model Driven Development (models as first-order citizens) and Software Product Lines (variability management).

Abstraction is one of the fundamental principles of software engineering in order to master complexity (Kramer, 2007). We make use of a model-driven approach to promote abstraction in self-adaptive systems development. The *self-adaptive systems community* has considered the

use of models to describe adaptation for long. Models allow the application of verification and validation methods during the development process and can support self-adaptation at run-time (Cheng et al., 2009). The *models@run-time community* is particularly concerned with this research area and uses model-driven techniques for validating and monitoring run-time behavior (Blair et al., 2009).

In a context where the possible combinations of situations are constantly increasing, the implementation of ad-hoc solutions to cover all possible variants is not feasible. DSPLs represent an efficient way for modeling adaptive capabilities in software systems (Hallsteinsen et al., 2008). A key characteristic in DSPLs is the intensive use of variability at run-time in order to adapt the system configuration caused by a context or an environment change.

The starting point of the proposed design method is a specification of a self-adaptive system made up of two parts:

1. **A variability model**, which provides an intensional description for each of the possible system configurations. From the different variability modeling techniques (e.g. Feature Modeling (Kang et al., 1990), UML profiles with stereotypes (Gomaa & Shin, 2008) etc.) in this thesis it is chosen Feature Modeling because it is the most popular technique in product line engineering. Specifically, a Feature Model (FM) whose features can be activated or deactivated at run-time. Each configuration consist of the features that are active in the FM.

   Feature Modeling is a technique to specify the variants of a system in terms of features (coarse-grained system functionality) (Czarnecki & Kim, 2005). Features are hierarchically linked in a tree-like structure through variability relationships. There are four relationships related to variability concepts:

   - *Optional.* A feature can be selected or not whenever its parent feature is selected. Graphically it is represented with a small white circle on top of the feature.
   - *Mandatory.* A feature must be selected whenever its parent

feature is selected. It is represented with a small black circle on top of the feature.

- *Or-relationship.* A set of child features have an or-relationship with their parent feature when one or more child features can be selected simultaneously. Graphically it is represented with a black triangle.
- *Alternative.* A set of child features have an alternative relationship with their parent feature when only one feature can be selected simultaneously. Graphically it is represented with a white triangle.

In addition, cross-tree constraints between features are allowed:

- *Require.* The selection of a feature implies the selection of another feature. Graphically it is represented with a one-way arrow.
- *Exclude.* The selection of a feature excludes the selection of another feature. Graphically it is represented with a two-way arrow).

2. **A set of resolutions**, which defines reconfigurations among the different system configurations in a declarative manner. Finding the optimal formalism for modeling adaptation policies remains an open research question (Cetina et al., 2009b). For illustration purposes, we propose using resolutions as a general concept to represent reconfigurations. A resolution (R) is a list of pairs (F,S) in which F indicates a feature in the FM and S the feature state (True or False depending on whether or not the feature is active). Each resolution is associated with a context condition and represents the sequence of actions in terms of feature activation/deactivation produced in the system when a context condition is fulfilled:

$$[\text{guard}]\ R = \{\ (\text{F, S})\ | F \in [FM] \wedge S \in \{\text{True, False}\}\ \}$$

Using resolutions, the reconfigurations must not be specified for every feature combination. Furthermore, resolutions may have

guards to constrain the configurations where can be executed. A guard is a Boolean expression that has to be evaluated before applying the resolution. For instance:

$$\left[!c2\right] R1_{condx} = \{(\text{F1,True}),\ (\text{F2,False})\}$$

This resolution indicates that when the context condition *condx* is fulfilled, feature F1 must be activated and F2 must be deactivated. The guard $\left[!c2\right]$ avoids the application of resolution $R1$ when the system is on configuration c2.

Typically, once the designer have modeled the system, the variability specification is used at run-time to drive the system autonomic behavior. Nevertheless, designers can insert invalid or undesirable behavior into the system unintentionally. In order for an adaptive program to be trusted, it is important to have mechanisms to ensure that the program functions correctly during and after adaptations (Zhang & Cheng, 2006). Therefore, we propose a design method to analyze and adjust variability specifications before using them at run-time.

4.1.1 Design Method Overview

In this section we provide an overview of the method proposed for designing self-adaptive systems.

Self-adaptive systems are driven by reconfigurations at run-time originated by context events. Reconfigurations can have a great number of repercussions and situations. Designers must usually deal with large-scale systems with a large number of possible configurations and reconfigurations among them. Given a variability specification that models and adaptive system, it is difficult to foresee the behavior that the system will perform at run-time. Therefore, we propose to extend the design time by applying the design method defined in this thesis. This new stage assures that systems will never reach inconsistent states or originate errors during execution. Figure 4.1 shows an overview of the typical self-adaptive system development approach (a) and the extended version presented in this master thesis (b). The design method

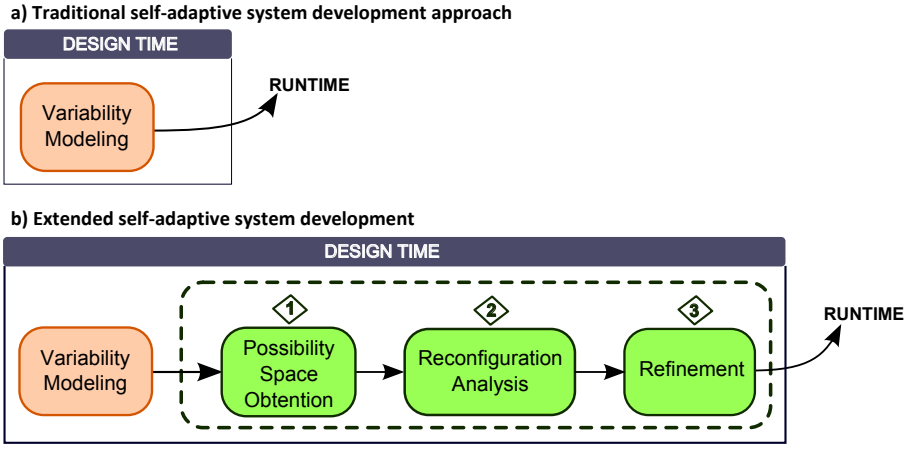


Figure 4.1: Stages involved in the design method for self-adaptive systems

involves three new sequential steps: (1) Possibility Space Obtention, (2) Reconfiguration Analysis, and (3) Variability Refinement.

The key steps to apply the design method are briefly introduced:

1. **Possibility Space Obtention.** In this step it is obtained a representation that eases the analysis of the system behavior. From the initial variability specification, a state machine model is derived. This state machine represents the Possibility Space of the system. The Possibility Space contains all the feasible configurations that the system can reach through execution, and the reconfigurations among them.

2. **Reconfiguration Analysis.** In the second step, making use of the representation obtained in previous step, the possible reconfigurations are analyzed with respect to different design properties identified.

3. **Variability Refinement.** Finally, in the last step, the system specification is refined in order to guarantee the design properties.

The design method is iterative. Steps 2 and 3 may be repeated as many times as necessary to assure different design properties. The system design is successively refined until it meets the requirements of the system application domain and/or the users. When no further changes are required, the system specification is used to guide the execution of the system at run-time.

4.2 Design Method

This section provides a more detailed description on the design method proposed in this master thesis.

We have defined a design method for self-adaptive systems for the purpose of analyzing the effects of run-time reconfigurations at design time and providing refinements to guarantee design properties before using the models at run-time. The next subsections provide further details on the steps involved in the design method.

4.2.1 Possibility Space Obtention

The first step in the design method is to obtain a representation that eases the understanding and analysis of the behavior that the system will perform at run-time. Figure 4.2 summarizes the first step in the design method.

The execution of an adaptive system can be abstracted as a state machine model (Bencomo et al., 2008c; Zhang & Cheng, 2006) where each state represents a distinct configuration of the same system and each transition represents a reconfiguration between configurations. This graphical representation offers an overall view of the whole process of run-time reconfiguration and reduces both effort and complexity during analysis.

Given a variability specification, made up of a feature model and a set of resolutions, we can derive a state machine model that represents the behavior of the system generated by that specification. We denote by Possibility Space that state machine model. The Possibility

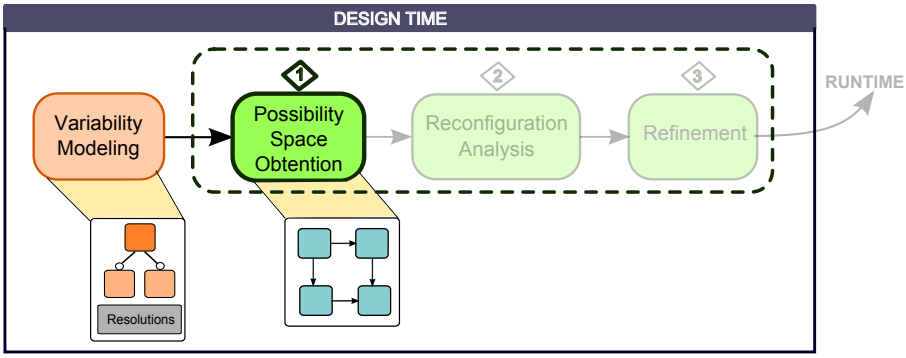


Figure 4.2: Overview of the first step in the design method

Space represents all feasible configurations that the system can reach because the fulfillment of each resolution. The **derivation process** works as follows: given an initial system configuration (active features in the FM), all the resolutions are applied to the initial configuration, obtaining new configurations. Then, for each new configuration, all the resolutions are applied again. The process is repeated until there are no new configurations generated.

Figure 4.3 shows an example of a Possibility Space (4.3b) derived from a variability specification (4.3a) that contains a variability model with four features (F1, F2, F3 and F4) and three resolutions (Rx, Ry and Rz). It can be observed how the definition of 3 resolutions generates a Possibility Space with 8 possible configurations and 12 reconfigurations among them. The number of configurations and transitions grows rapidly; the number of configurations explodes in a combinatorial way with regard to the number of variants, and the number of transitions is quadratic with regard to the number of configurations (Morin et al., 2009).

In the case of large systems, the specification of adaptive systems involves great complexity and becomes a tedious task prone to errors. Therefore, it would be interesting to have design methods and tools to assist designers in the definition of this kind of systems. The state

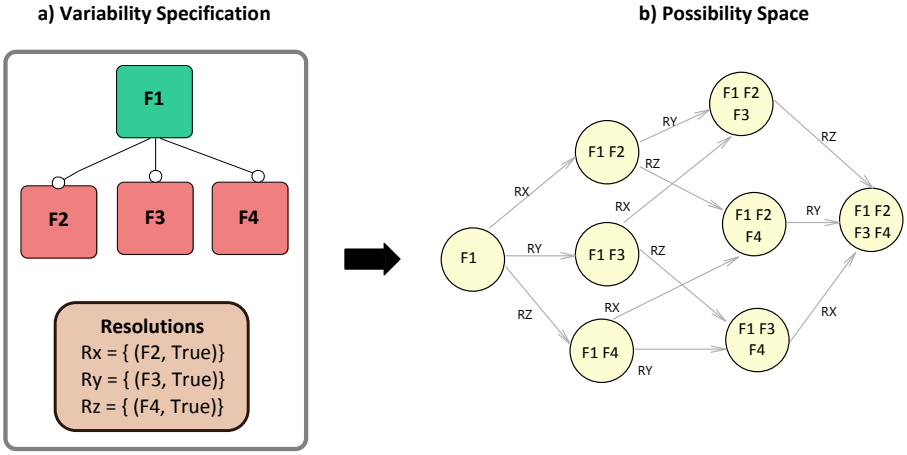


Figure 4.3: Obtaining the Possibility Space from a variability specification

machine representation eases the visualization of the system behavior and lets designers perform extensive validation of the system's dynamic variability before its execution.

4.2.2 Reconfiguration Analysis

Once the Possibility Space has been obtained, the following step is to analyze the reconfiguration effects. Figure 4.4 shows an overall view of the second step in the design method.

The reconfiguration analysis is based on the STAIRS approach (Haugen et al., 2005; Haugen & Sttølen, 2003), that provides a foundation for specification analysis, verification, and testing. STAIRS defines an incremental system development based on refinements. STAIRS focuses on refinement, which is a development step where the specification is made more complete by information being added to it, in such a way that any valid implementation of the refined specification will also be a valid implementation of the original specification (Runde et al., 2005).

Although STAIRS is applied to analyze UML interactions by means

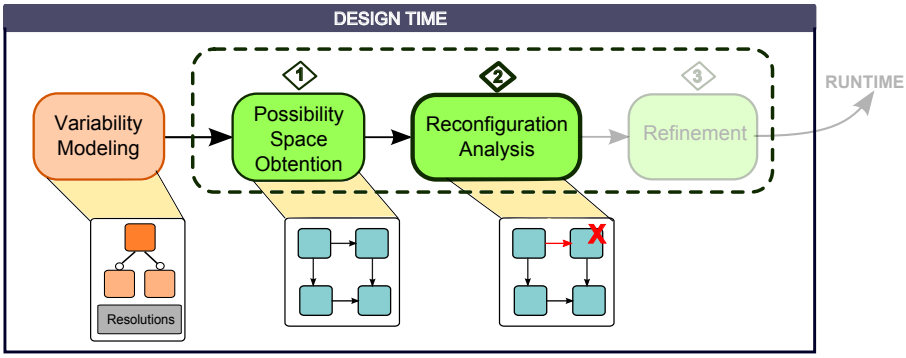


Figure 4.4: Overview of the second step of the design method

of sequence diagrams, the approach can be applied to other types of behaviors (Haugen et al., 2005). We apply STAIRS in behavior specifications in the form of state machines. In state machines, the behavior of the system is described as transitions. In our case the transitions represent feasible system reconfigurations. We denote R as the set of all the reconfigurations in the Possibility Space. Following STAIRS, a reconfiguration can be categorized as:

- **positive**, meaning that it is valid, legal, or desirable;
- **negative**, meaning that it is invalid, illegal, or undesirable;
- **inconclusive**, meaning that it is considered irrelevant.

In our approach, this categorization corresponds with the result of checking interesting design properties of the Possibility Space. We have proposed a series of desirable design properties (such as Redundancy, Reversibility, Determinism etc.) based on previously established design issues. The catalog of properties is presented in detail in chapter 5.

The process to follow in this step is as follows: initially, we assume that all the reconfigurations in R are positive because the designer has specified the desirable behavior of the system. Nevertheless, designers can define undesirable or illegal behavior unintentionally; thus the

reconfiguration effects should be evaluated. In the Reconfiguration Analysis step, the designer selects a design property to check if the reconfigurations fulfill determined assertions about design behavior defined by the property. Accordingly, the set R is subdivided into two subsets $R+$ and $R-$ depending on whether or not the reconfigurations satisfy the property. $R+$ contains the positive behavior (reconfigurations that fulfill the property) and $R-$ contains the negative behavior (reconfigurations that violate the property). The negative behavior ($R-$) should be avoided from the specification. For each property, we have defined a refinement to detect the reconfigurations that violate the property and keep them out of the specification. Next step is in charge of applying the corresponding refinements to avoid the negative behavior.

4.2.3 Variability Refinement

The last step in the design method is to refine the specification in order to avoid the negative behavior. Figure 4.5 summarizes the last step in the design method.

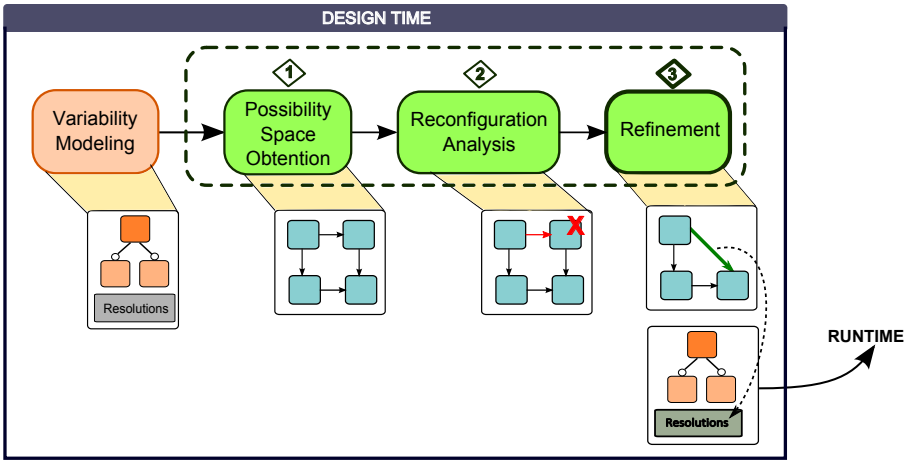


Figure 4.5: Overview of the last step of the design method

The purpose of the refinement is to reduce the allowed behavior to capture new design decisions or to match the problem more adequately. The refinement is responsible for avoiding negative behavior from the specification guaranteeing the invariant: **all original positive behavior remains positive**. That means eliminating $R-$ from the specification but keeping $R+$ intact. This invariant is very important because we do not want to remove negative behavior at the expense of losing reconfiguration capabilities. In this sense, the refinement does not remove resolutions. It only ignores negative reconfigurations from the specification by adding guards to the resolutions that trigger negative reconfigurations. The guards constrain the possibilities of reconfiguration, avoiding the execution of reconfigurations in undesirable configurations. Moreover, the refinement can add new reconfigurations to fulfill a property. The steps 2 and 3 can be successively repeated to ensure different properties. Chapter 5 presents the design properties defined and provides detailed guidelines to perform the associated refinements.

4.3 Conclusions

This chapter introduces a systematic design method for self-adaptive systems based on refinements. The explosion of configurations is one of the main problems when designing self-adaptive systems. The high-number of possible configurations and reconfigurations might lead to several undesirable consequences during execution. Therefore, the main aim of the design method is to analyze and to adjust variability specifications in order to guarantee trustworthy adaptations at run-time.

The design method involves three sequential steps: (1) Possibility Space Obtention, (2) Reconfiguration Analysis, and (3) Variability Refinement. Summarizing, first the behavior of the system is analyzed before run-time with respect to different design properties. We have defined a series of refinements in order to force the system to fulfill each property automatically. The refinements always keep the invariant that all original valid reconfigurations remain valid, so that reconfiguration capacity is not lost. The design method is an iterative process that

may be repeated systematically to ensure different properties through refinements.

Chapter 5

Design Properties Catalog

In this chapter, we term a series of desirable design properties for self-adaptive systems and particular refinements to assure each property automatically.

Depending on the application domain of the system, designers can chose the adequate properties that the system should satisfy throughout its execution. Dealing with these properties is essential to achieve reliably self-adaptive systems that satisfy the requirements of a particular domain. These properties are only an example; designers may include as many properties as they consider to cover different aspects of design and execution of self-adaptive systems. All the properties have been successfully applied in a case study presented in Chapter 6.

Figure 5.1 summarizes the design properties in form of a feature model. We have identified the following design properties:

1. *Safe Reconfigurations and Reachability.* This property guarantees that after reconfigurations the system never reaches invalid states. Invalid states refer to inconsistent configurations where variability

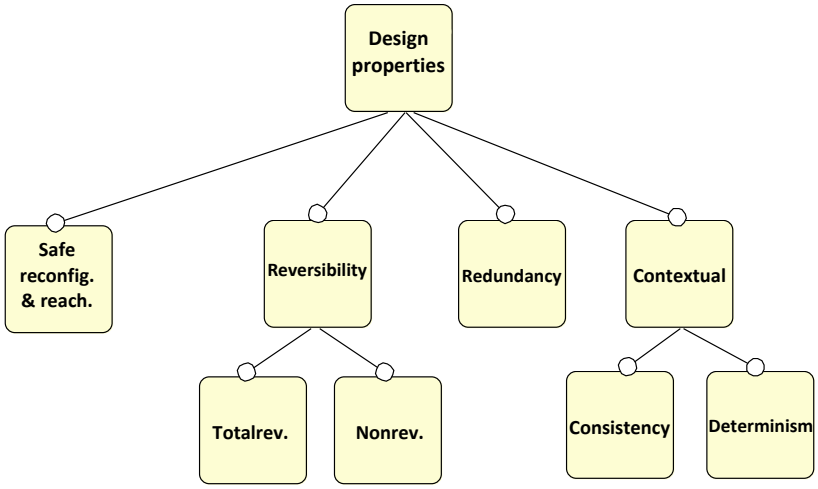


Figure 5.1: Design properties catalog

constraints are violated. The refinement associated to this property guarantees that all reconfigurations are safe (do not lead to invalid states) and that no configuration is reached through invalid states.

2. *Redundancy.* This property guarantees that the specification does not contain duplicated behavior. Redundancy is generally regarded as an undesirable property of model specifications. Redundancy implies designs or implementations of low quality and higher resource consumption.

3. *Reversibility.* In the context of adaptive systems where the reconfigurations are driven by the environment and the system is not under the users control, it might be interesting to add rollback capabilities enabling the system to reverse the effect of unexpected reconfigurations. This property involves two refinements:

   (a) *Total Reversible System*: For each reconfiguration, there is another reconfiguration that leads directly to the source configuration.

(b) *Nonreversible System*: For each reconfiguration, there is not another reconfiguration that leads directly to the source configuration.

4. *Contextual Consistency.* It could be desirable for the same action or set of actions to always produce the same effect in the system, independently of the current system state. This property guarantees that when a determined context condition is fulfilled, the system evolves (independently from its current state) to a predefined or controlled configuration.

5. *Contextual Determinism.* This property involves predictability in the reconfiguration process to future states. The refinement associated to this property assures that for each possible context condition, every state in the system has exactly one reconfiguration that leads to the next state. Thus, from a given state, when a determined context condition is fulfilled, the system can only reconfigure to one destination state.

The rest of the chapter is organized as follows. Section 5.1 presents the *Safe Reconfigurations and Reachability property* and the associated refinement to achieve systems free on unsafe reconfigurations and unsafely reachable configurations. Section 5.2 introduces the *Redundancy property* and the refinement to guarantee systems without duplicated behavior. Section 5.3 explains the *Reversibility property* and its two associated refinements to achieve either a Total Reversible or a Nonreversible system. Next, Section 5.4 describes the *Contextual Consistency property* and its associated refinement. Next, Section 5.5 details the *Contextual Determinism property* and the refinement to achieve deterministic systems. Finally, Section 5.7 concludes the chapter.

5.1 Safe Reconfigurations and Reachability Property

In this section, the Safe Reconfigurations and Reachability property is explained in detail.

5.1.1 Definition

As explained in previous section, the behavior of the system is driven by a set of resolutions. Resolutions are triggered by the fulfillment of context conditions. The combination of resolutions can originate inconsistent configurations, these are configurations where variability constraints are violated. The aim of this property is to guarantee that systems will never reach invalid configurations during execution. To check that the defined resolution set is actually within the variability model constraints, developers can apply existing feature model analysis tools. There are a number of tools supporting these analysis capabilities such as AHEAD Tool Suite (Ahead, 2009), FaMa Framework (Trinidad et al., 2008), Feature Model Plug-in (Plugin, 2009) and pure::variants (Pure, 2009). In this master thesis we have used the Feature Model Analyzer FaMa because is an open source project. FaMa validates a partial configuration consistency with a given feature model. Further, FaMa can combine multiple resolutions to ensure that there are no invalid configurations in a given situation.

Figure 5.2 shows a specification made up of (1) *a Feature Model (FM)* with 3 features: F1, F2 and F3, where only feature F1 is active; and a variability constraint of type "require" between features F2 and F3. (2) *A set of resolutions*, $R1_{cond1}$ and $R2_{cond2}$, expressing the actions to carry out (activate feature F2 and activate feature F3 respectively) when context conditions $cond1$ and $cond2$ are fulfilled. The FM and the set of resolutions define a Possibility Space composed by 4 different configurations ($S1$, $S2$, $S3$, $S4$). It can be observed that eventually, when the system is on configuration $S1$ and the context condition $condx$ is fulfilled, the resolution $R1$ is triggered. $R1$ activates feature F2 evolving the system to configuration $S2$. $S2$ is an invalid configuration since feature F2 is active but feature F3 is inactive, therefore the variability constraint ("F2 requires F3") is violated.

In order to avoid inconsistent configurations during execution, a refinement has been defined.

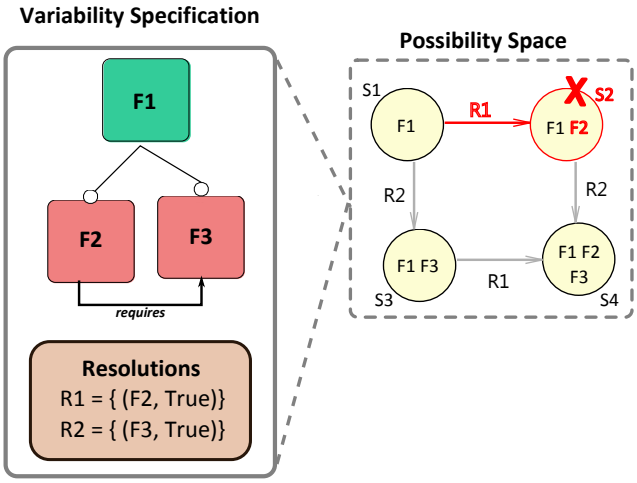


Figure 5.2: Safe Reconfigurations and Reachability property

5.1.2 Safe Reconfigurations and Reachability Refinement

This refinement is in charge of guaranteing safe reconfigurations during the whole system execution. The aim of the refinement is twofold:

1. *Avoid unsafe reconfigurations.* We consider unsafe reconfigurations the ones that evolve the system from a valid configuration to an invalid configuration. These reconfigurations are undesirable and have to be eliminated from the specification.

2. *Avoid unsafely reachable configurations.* When some reconfigurations are removed other configurations can be unreachable. In order to not loose reconfiguration capability, the refinement guarantees that all the configurations are safely reachable.

The pseudo-code of the refinement is the following:

Algorithm 1 Safe Reconfigurations and Reachability Refinement

Require: a set S = {$S1$, ... , Sn} ∈ PS and a set R= {$R1$, ... , Rn} of resolutions

Ensure: S without invalid configurations and unsafely unreachable configurations; R without unsafe reconfigurations.

```
1: for all s ∈ S do
2:    R_out ⇐ reconfigurations triggered from s
3:    for all Ri ∈ R_out do
4:       sr ⇐ configuration reachable by Ri
5:       if sr is invalid then
6:          Remove Ri by modifying the guard: [!s] Ri
7:          R_out−sr ⇐ reconfigurations triggered from sr
8:          for all Rj ∈ R_out−sr do
9:             Create a new resolution: [sr] Ri + Rj
10:         end for
11:      end if
12:   end for
13: end for
```

Starting from the initial configuration, the refinement iteratively checks the reconfigurations triggered in each configuration of the Possibility Space. If some reconfiguration reaches an invalid configuration, that reconfiguration is removed. The refinement introduces guards to those resolutions that trigger unsafe reconfigurations in order to avoid them from the specification. When some reconfigurations are removed could happen that other valid configurations are unreachable. Since the old unsafe reconfiguration lead to the current unreachable configuration, the refinement calculates a new safe reconfiguration which directly leads to the unreachable configuration from the valid source of the old unsafe reconfiguration. These new resolutions are calculated as the sequential composition of actions of the old unsafe resolution and each of the resolutions triggered from the invalid configuration. The condition of the new resolution is set as the conjunction of conditions of the resolutions implied in the composition. It also introduces guards to the new resol-

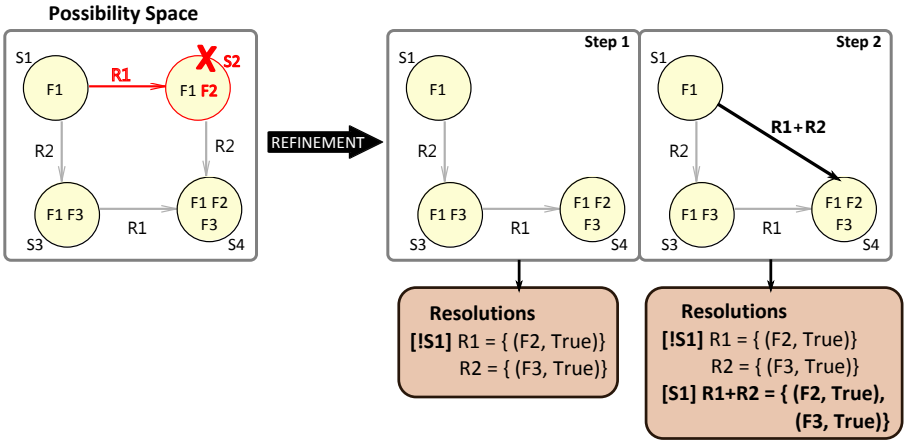


Figure 5.3: Safe Reconfigurations and Reachability refinement

ution in order to avoid the generation of new configurations because of the application of this resolution from other configurations.

Figure 5.3 illustrates the application of the refinement to the specification shown in Figure 5.2. In configuration $S1$ the reconfiguration triggered by $R1$ is unsafe because evolves the system to an invalid configuration ($S2$). Then, the refinement avoids the execution of resolution $R1$ in configuration $S1$ by modifying its guard. Next, in order to not loose reconfiguration capabilities, the refinement generates a new resolution from $S1$ to $S4$, as the composition of the previous resolutions $R1$ and $R2$. The guard of this new resolution is set to enable its execution only in configuration $S1$. The resolutions are modified as follows:

$$[!S1]R1_{cond1} = \{(\text{F2,True})\}$$
$$[S1]R1{+}R2_{cond1 \wedge cond2} = \{(\text{F2,True}),\ (\text{F3},\ \text{True})\}$$

5.2 Redundancy Property

In this section, the Redundancy property is explained in detail.

According to (Bellotti & Edwards, 2001), simplicity is highly appreciated by users in context-aware systems (as is the case for self-adaptive systems). Furthermore, as stated in (Dey, 2009), one of the biggest challenges to the usability of context-aware applications is the difficulty that users have to understand what a system is doing. Redundancy property reduces the number of reconfigurations by avoiding duplicated behavior, thus improving both simplicity and understanding of the system behavior.

5.2.1 Definition

We understand as redundancy to be the duplication of behavior. Two or more different resolutions are redundant if they produce the same effect on the system. That is to say, they evolve the system from the same source configuration to the same target configuration by means of different actions. Since these reconfigurations are wasteful of resources (such as storage and execution time), lead to inconsistent specifications, and make maintenance more complex, they should be eliminated. The elimination of redundancy optimizes design and can actually improve execution time and understandability of the system behavior (Nance et al., 1999).

For example, given a FM with four features (F1, F2, F3 and F4), with F1 active, a designer defines the following set of resolutions:

$R1_{cond1}$ ={(F2,True), (F3,True)}
$R2_{cond2}$ ={(F4,True), (F2,False), (F3,True),(F4,False),(F2,True)}
$R3_{cond3}$ ={(F2,True)}
$R4_{cond4}$ ={(F4,True)}
$R5_{cond5}$ ={(F4,False)}

The above resolutions generate the Possibility Space illustrated in Figure 5.4. It can be observed that the Possibility Space contains two redundant resolutions $R1$ and $R2$ because both evolve the system between the same pairs of configurations: $S1$-$S2$ and $S3$-$S2$. Therefore, a refinement can be applied to avoid redundant resolutions and improve the system specification and execution.

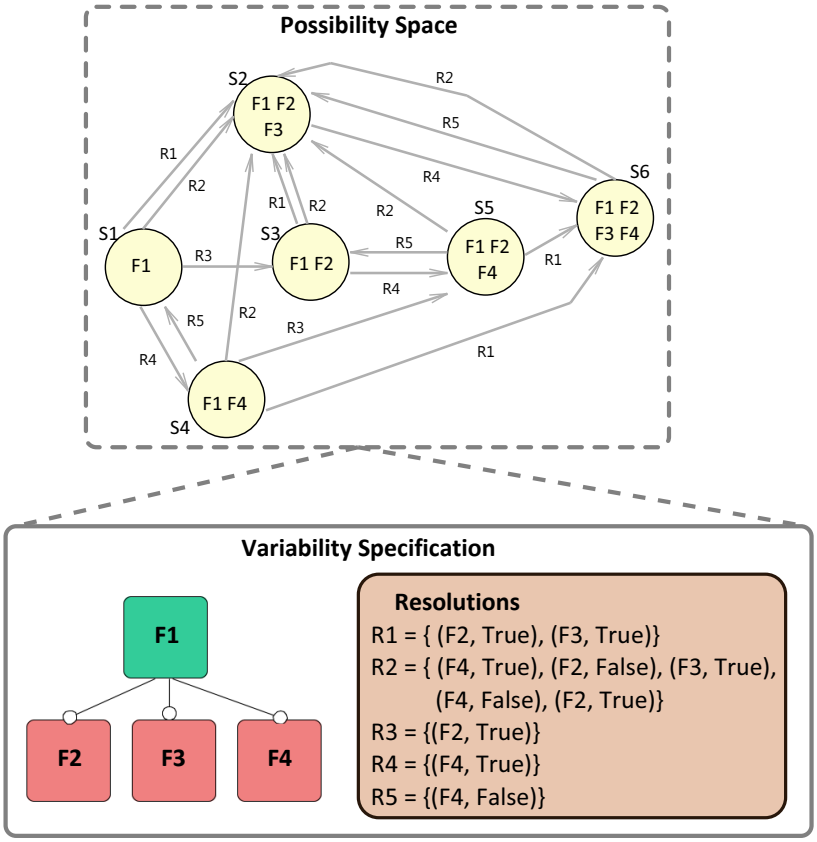


Figure 5.4: Redundancy Property

5.2.2 Redundancy Avoidance Refinement

This refinement eliminates redundant reconfigurations in a self-adaptive system specification. By reducing the number of duplicated reconfigurations, this refinement improves simplicity and, consequently understanding of the system execution. The refinement, first finds redundant resolutions. Then, it selects the simplest resolution and removes the others. We consider the simplest resolution as the one that involves the minimum number of change actions. Finally, in order to guarantee the

invariant eliminating negative behavior without altering positive behavior, the condition and guard of the simplest resolution are modified to include the possibilities of reconfiguration of the removed redundant resolutions.

The pseudo-code of the refinement is the following:

Algorithm 2 Redundancy Avoidance Refinement

Require: a set R= {$R1$, ... , Rn} of resolutions
Ensure: R' without redundant resolutions
1: **for all** $Ri \in R$ **do**
2: Find the redundant resolutions of Ri,
$R- \Leftarrow$ redundant resolutions of Ri
3: Select the simplest resolution $Rs \in R-$
4: **for all** $Rj \in R-/\ Rj \neq Rs$ **do**
5: Modify the condition of Rs: $cond(Rs) = cond(Rs) \vee cond(Rj)$
6: Modify the guard of Rs: $guard(Rs) = guard(Rs) \vee guard(Rj)$
7: Remove Rj
8: **end for**
9: **end for**

The above Possibility Space (Figure 5.4), contains two redundant resolutions: $R1$ and $R2$. $R1$ implies two actions ({(F2,T), (F3,T)}) whereas $R2$ implies five actions ({(F4,T), (F2,F), (F3,T), (F4,F), (F2,T)}). Therefore, the refinement eliminates $R2$ and modifies the guard and condition of $R1$ in order to not loose reconfiguration possibilities. The refinement modifies the set of resolutions as follows:

$$
\begin{aligned}
& R1_{cond1 \vee cond2} = \{(\text{F2,True}),\ (\text{F3,True})\} \\
[!S1; !S3]\ & R2_{cond2} = \{(\text{F4,True}),\ (\text{F2,False}),\ (\text{F3,True}), \\
& \qquad (\text{F4,False}),(\text{F2,True})\} \\
& R3_{cond3} = \{(\text{F2,True})\} \\
& R4_{cond4} = \{(\text{F4,True})\} \\
& R5_{cond5} = \{(\text{F4,False})\}
\end{aligned}
$$

The resulting Possibility Space is shown in Figure 5.5. The removed resolutions by the refinement are represented with red dashed lines.

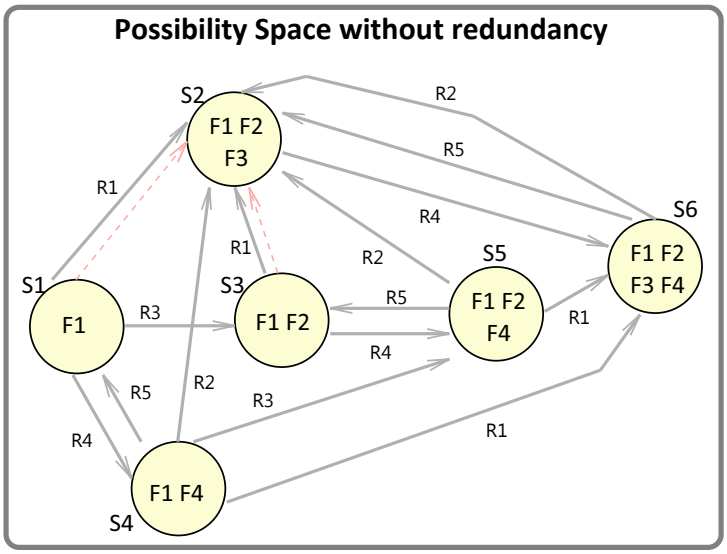


Figure 5.5: Redundancy Avoidance Refinement

5.3 Reversibility Property

In this section, the Reversibility property and its associated refinements are explained in detail.

As stated in (Cetina et al., 2010), recovering from a failed reconfiguration is a key design issue in DSPLs in making users feel more comfortable with systems. Designing systems with rollback capabilities is a daunting task especially in the case of large systems. This property automatically guarantees rollback capabilities to reconfigurations.

5.3.1 Definition

An interesting concern in self-adaptive system execution is recovery from an undesired or unexpected reconfiguration (Cetina et al., 2010). This property allows reconfigurations to be undone and restore past states. We define reversibility as the possibility to return to a previous state after a reconfiguration has been applied. If there exists a

reconfiguration $R1$ that evolves the system from configuration $S1$ to $S2$ and there exists another reconfiguration that returns the system from $S2$ to $S1$, the reconfigurations are said to be reversible. Based on this property we classify systems under two main categories:

- *"Total Reversible System"*: for all reconfigurations occurring in the system, there exists a reversible reconfiguration.
- *"Nonreversible System"*: after each reconfiguration, the system cannot be restored in its previous configuration.

For example, given a FM with four features F1, F2, F3 and F4, where F1 is active, a designer defines the following set of resolutions:

$$R1_{cond1} = \{(\text{F2,True}), (\text{F3,True})\}$$
$$R2_{cond2} = \{(\text{F2,True})\}$$
$$R3_{cond3} = \{(\text{F4,True})\}$$
$$R4_{cond4} = \{(\text{F4,False})\}$$

The above resolutions generate the Possibility Space illustrated in Figure 5.6. It can be observed that the Possibility Space contains reversible reconfigurations between configurations S1-S4, S2-S6, and S3-S5. These reconfigurations are triggered by resolutions $R3$ and $R4$. In order to deal with reversibility two different refinements have been defined.

5.3.2 Reversibility Refinement

Depending on the type of reversible system that is required, the designer can apply one of the following refinements making either a *Total Reversible System* or a *Nonreversible System*.

5.3.2.1 Total Reversible System Refinement

The purpose of this refinement is to ensure that, for all configurations contained in the Possibility Space, there will exist a reconfiguration that leads directly to the previous configuration. The refinement generates

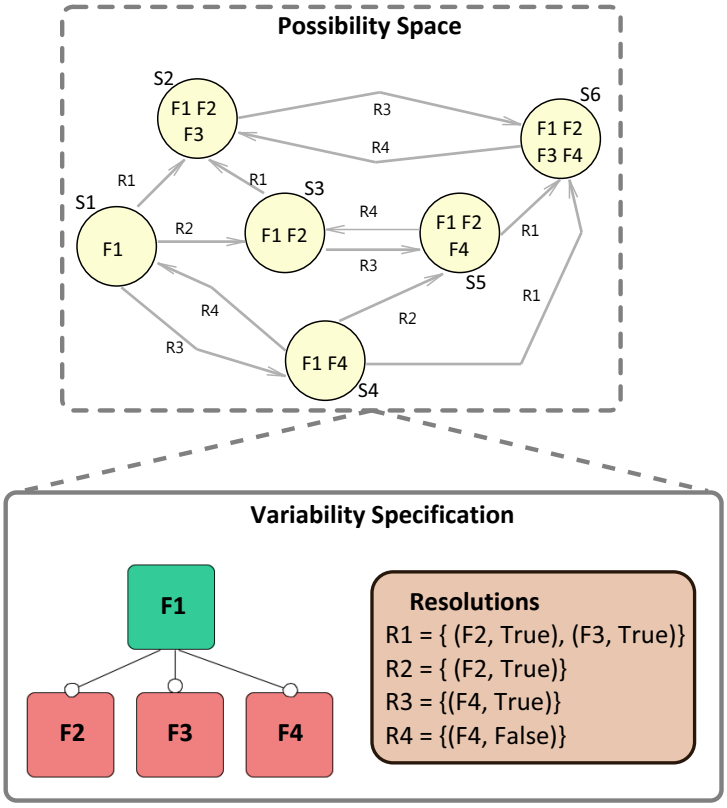


Figure 5.6: Reversibility Property

new reconfigurations that assure that every reconfiguration has a reverse reconfiguration. The new reconfigurations define compensation actions to reverse a reconfiguration. The compensation actions are calculated by means of the *Relative Complement* operator (\) which is also known as the set-theoretic difference. Given a resolution Ri that evolves the system from configuration C_{source} to configuration $C_{destination}$, the reversible reconfiguration $\overline{Ri}$ (that evolves system from $C_{destination}$ to C_{source}) is calculated as follows: the features that are active in the destination configuration and are inactive in the source configuration should be deactivated. This set of features is denoted *Deactivate*. The

features that are active in the source configuration and are inactive in the destination configuration should be activated. This set of features is denoted *Activate*. The refinement introduces guards to the new generated resolutions to avoid the generation of new configurations because of the application of this resolution from other configurations.

The pseudo-code of the refinement is the following:

Algorithm 3 Total Reversible System Refinement

Require: a set R= $\{R1, \ldots, Rn\}$ of resolutions
Ensure: a set R' with reversible resolutions
1: **for all** $Ri \in R$ **do**
2: Calculate the features to deactivate:
Deactivate= $\{(F,S) \in C_{dest} | S = True\} \setminus$
$\{(F,S) \in C_{source} | S = True\}$
3: Calculate the features to activate:
Activate= $\{(F,S) \in C_{source} | S = True\} \setminus$
$\{(F,S) \in C_{dest} | S = True\}$
4: Add a new resolution $\overline{Ri} = Activate \cup \overline{Deactivate}$
5: Set condition and guard of the new resolution
6: **end for**

Given the Possibility Space of Figure 5.6, it can be observed that the system is not totally reversible because there exist reconfigurations that do not have a reverse reconfiguration (the reconfigurations triggered by resolutions $R1$ and $R2$). The designer may apply the previous refinement algorithm to obtain a *Total Reversible System*. The refinement modifies the set of resolutions as follows:

$[S2; S6]$ $\overline{R1}_{cond5}$={(F2,False), (F3,False)}
$[S2]$ $\overline{R1}_{cond6}$={(F3,False)}
$[S3; S5]$ $\overline{R2}_{cond7}$={(F2,False)}

Figure 5.7 shows the Possibility Space after applying this Total Reversible refinement. The generated resolutions are represented with thick lines.

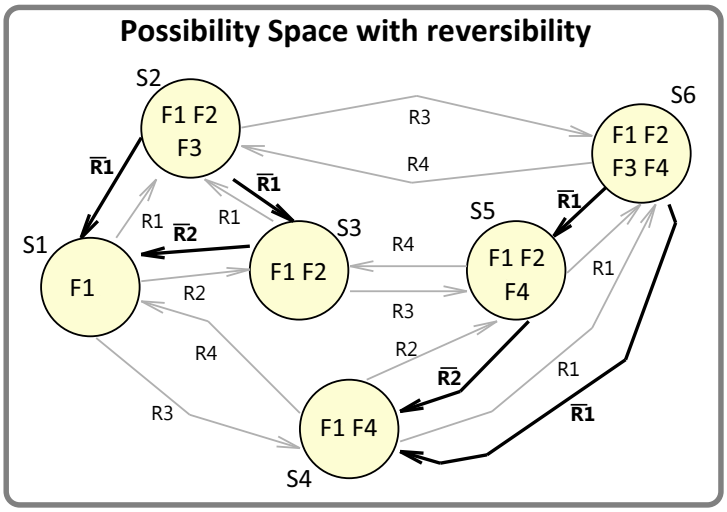


Figure 5.7: Total Reversible System Refinement

5.3.2.2 Nonreversible System Refinement

The purpose of this refinement is to ensure that there is no reconfiguration that leads directly to the previous configuration. For each pair of reversible reconfigurations, the refinement removes one of the two reconfigurations. In order to keep the invariant *don't lose positive behavior*, those resolutions are not actually removed. The refinement modifies their guards to constrain when those reconfigurations can be executed. The pseudo-code of the refinement is the following:

Algorithm 4 Nonreversible System Refinement

Require: a set R= {$R1$, ... , Rn} of resolutions
Ensure: R without reversible resolutions
1: **for all** $Ri \in R$ **do**
2: Find a reversible resolution of Ri,
$RR \leftarrow$ reversible resolution of Ri
3: **if** $RR \neq \emptyset$ **then**
4: Remove Ri or RR by modifying the guard
5: **end if**
6: **end for**

Given the Possibility Space of Figure 5.6, it can be observed that the space contains reversible reconfigurations between configurations S1-S4, S2-S6, and S3-S5. These reconfigurations are triggered by resolutions $R3$ and $R4$. If the designer requires a *Nonreversible System*, the *Nonreversible System refinement* can be applied to achieve it. Then the refinement removes resolution $R3$ or $R4$. The designer can choose which of these two resolutions should be removed. Consequently, two different Possibility Spaces can be obtained:

- If the designer decides to remove $R3$, the refinement modifies the guard of the resolution $R3$ to be avoided from the specification:

$$\left[!S1; !S2; !S3\right]\ R3_{cond3} = \{(\text{F4,True})\}$$

This guard avoids the application of resolution $R3$ when the system is on configuration S1, S2 or S3. The Possibility Space after applying the refinement is shown in Figure 5.8 (left).

- If the designer decides to remove $R4$, the refinement modifies the guard of the resolution $R4$ to be avoided from the specification:

$$\left[!S4; !S5; !S6\right]\ R4_{cond4} = \{(\text{F4,False})\}$$

This guard avoids the application of resolution $R4$ when the system is on configuration S4, S5 or S6. The Possibility Space after applying the refinement is shown in Figure 5.8 (right).

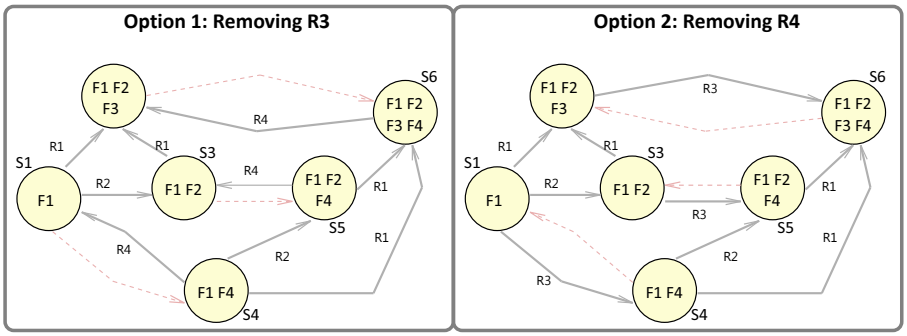


Figure 5.8: Feasible Possibility Spaces after applying the *Nonreversible System* refinement

The choice between alternatives has no implications initially and depends only on the designer's decisions.

5.4 Contextual Consistency Property

In this section the Contextual Consistency property is explained in detail.

5.4.1 Definition

This property guarantees that when a determined context condition is fulfilled, the system evolves (independently from its current state) to a predefined or controlled configuration.

If the designer wants that when an exceptional context condition is fulfilled the system always reconfigures to a new configuration, he would have to foresee the needed reconfigurations and to specify them manually. As the number of possible configurations increases, this task is becoming tedious and prone to errors. Therefore, we have defined a refinement to generate all the needed reconfigurations automatically.

Given a FM with four features (F1, F2, F3 and F4), where F1 is

active, a designer defines the following set of resolutions:

$$R1_{cond1} = \{(\text{F2,True})\}$$
$$R2_{cond2} = \{(\text{F3,True})\}$$

The above resolutions generate the Possibility Space illustrated in Figure 5.9.

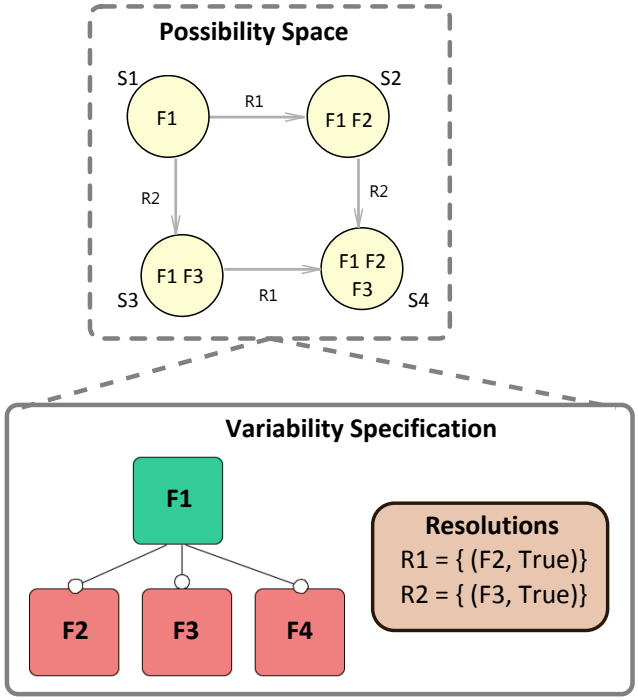


Figure 5.9: Contextual Consistency Property

5.4.2 Contextual Consistency Refinement

This refinement generates from each configuration in the Possibility Space a new reconfiguration that evolves the system to a new configuration (S_{new}) when a context condition $cond_{exc}$ is fulfilled.

The pseudo-code of the refinement is the following:

Algorithm 5 Contextual Consistency Refinement

Require: a set S = $\{S1, \ldots, Sn\} \in$ PS, a set R= $\{R1, \ldots, Rn\}$ of resolutions and a new configuration s_{new}

Ensure: R containing resolutions from every configuration $\in$ PS to s_{new}

1: **for all** s $\in$ S **do**
2: Add a new resolution R_{new} from s to s_{new}
3: Calculate the features to activate:
Activate= $\{(F, S) \in s_{new} | S = True\} \setminus$
$\{(F, S) \in s | S = True\}$
4: Calculate the features to deactivate:
Deactivate= $\{(F, S) \in s | S = True\} \setminus$
$\{(F, S) \in s_{new} | S = True\}$
5: $[s]R_{exc} = Activate \cup \overline{Deactivate}$
6: **end for**
7: Add a new resolution from s_{new} to the *initial configuration* of the system

The refinement generates new resolutions that trigger reconfigurations from each configuration in the Possibility Space to the new predefined configuration. The set of reconfiguration actions to reach the new configuration from each existing configuration is calculated as the union of two subsets: (1) Activate (features that are active in the new configuration and are inactive in the source configuration); (2) Deactivate (features that are inactive in the new configuration and are active in the source configuration). The features of these two subsets are calculated by the *Relative Complement* operator ($\setminus$). The guard of the new resolutions is set in order to trigger the resolutions only in the adequate configurations. Furthermore, a new resolution from the destination configuration to the initial configuration of the system is created. This resolution initializes the reconfiguration process after reaching the new exceptional configuration. The reconfiguration actions of this last resolution are calculated in the same way that the other resolutions

generated by the refinement.

Figure 5.10 shows the Possibility Space after applying the refinement to the specification illustrated in Figure 5.9. The new resolutions are represented with thick lines.

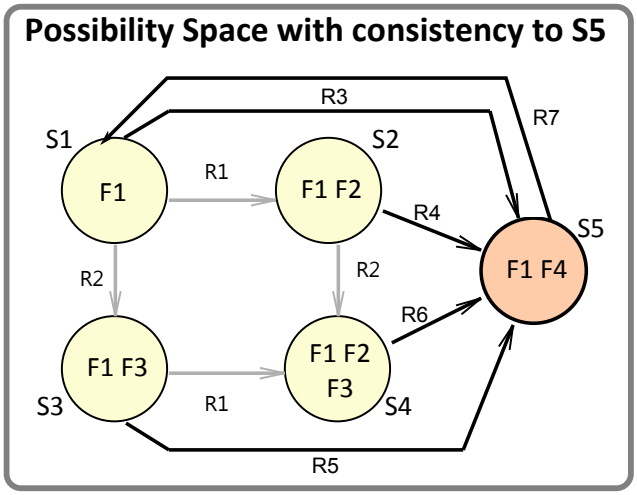


Figure 5.10: Contextual Consistency Refinement

The refinement generates the following new resolutions:

$[s1]$ $R3_{cond_{exc}}$={(F4,True)}
$[s2]$ $R4_{cond_{exc}}$={(F2,False), (F4, True)}
$[s3]$ $R5_{cond_{exc}}$={(F3,False), (F4, True)}
$[s4]$ $R6_{cond_{exc}}$={(F2,False), (F3, False), (F4, True)}
$[s5]$ $R7_{cond_{!exc}}$={(F4, False)}

5.5 Contextual Determinism Property

In this section, the Contextual Determinism property is explained in detail.

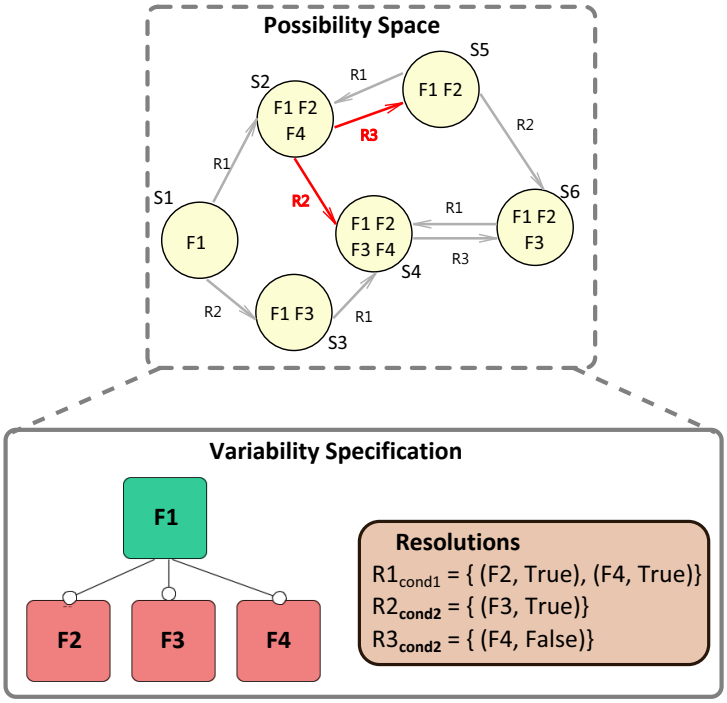


Figure 5.11: Contextual Determinism Property

5.5.1 Definition

A system is determinist if in each state in which the execution can follow different reconfigurations, conditions on those multiple reconfigurations are different. If two conditions would be satisfiable simultaneously then two different reconfigurations of the system would be carried out concurrently and the system will reach an inconsistent state. Therefore, non-determinist behavior is unacceptable in this kind of systems. Summarizing, a system is determinist if between the same pair of states two (or more) different reconfigurations cannot be triggered at the same time.

Given a FM with four features (F1, F2, F3 and F4), with F1 active, a designer defines the following set of resolutions:

$R1_{cond1}$ = {(F2,True), (F4, True)}
$R2_{cond2}$ = {(F3,True)}
$R3_{cond2}$ = {(F4,False)}

The above resolutions generate the Possibility Space illustrated in Figure 5.11. It can be observed that the system is non-determinist since in configuration $S2$ when the context condition *cond2* is fulfilled, resolutions $R2$ and $R3$ are triggered simultaneously. Then the system has to evolve to configurations $S4$ and confc5 at the same time, reaching an inconsistent state. We have defined a refinement to guarantee that the behavior of the system is always determinist.

5.5.2 Contextual Determinism Refinement

This refinement checks if the resolutions will generate non-determinist behavior during execution. In that case, the refinement modifies the set of resolutions in order to avoid the simultaneous reconfigurations.

The pseudo-code of the refinement is the following:

Algorithm 6 Contextual Determinism Refinement

Require: a set S = {$S1$, ... , Sn} ∈ PS and a set R= {$R1$, ... , Rn} of resolutions
Ensure: a deterministic PS

```
1: for all s ∈ S do
2:    R_out ⇐ reconfigurations triggered from s
3:    for all Ri ∈ R_out do
4:       Cond ⇐ associated conditions to the resolutions triggered by s
5:       if Cond contains duplicated conditions then
6:          Change one of the duplicated conditions
            or
7:          Eliminate reconfigurations associated to the duplicated conditions
8:       end if
9:    end for
10: end for
```

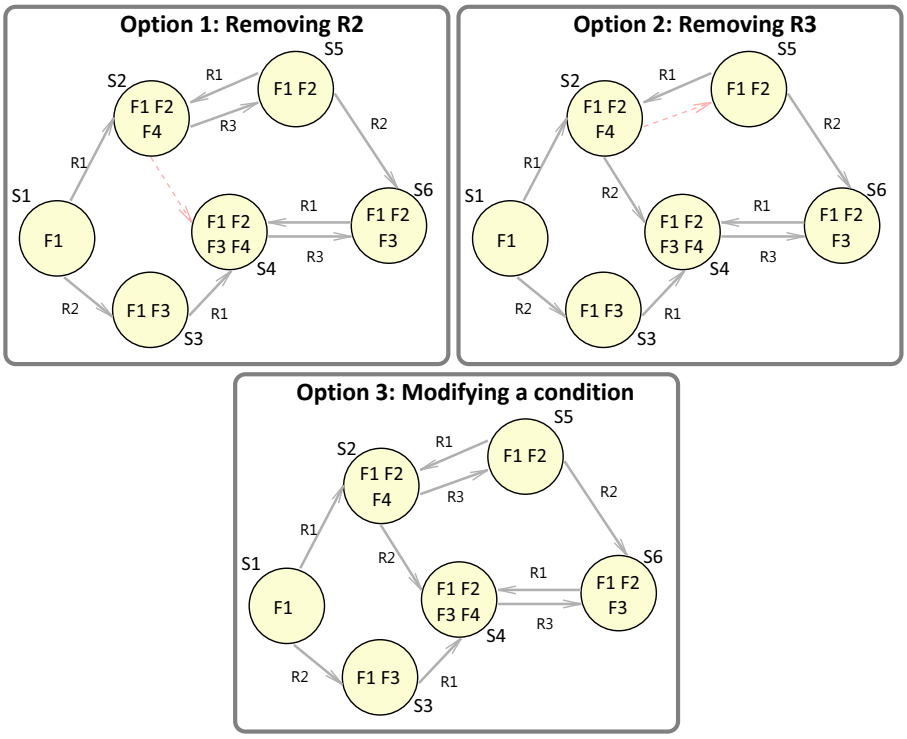


Figure 5.12: Contextual Determinism Refinement

For each state checks the set of context conditions that trigger reconfigurations to different destination states. If this set contains duplicate conditions the system is non-deterministic. The refinement can achieve a deterministic system in two different ways:

1. Removing simultaneous reconfigurations (keeping only one of them).
2. Modifying the condition of some resolutions in order to avoid simultaneous reconfigurations.

As illustrated in Figure 5.11, the system is non-determinist from configuration $S2$ (resolutions $R2$ and $R3$ are triggered simultaneously

when the context condition *cond2* is fulfilled). The refinement offers different alternatives to achieve a determinist system. The designer can select which alternative has to be carried out. Consequently, different Possibility Spaces can be obtained:

- If the designer decides to remove $R2$. The refinement modifies the guard of the resolution $R2$ to avoid its execution on configuration $S2$:

$$\left[!s2\right]\ R2_{cond2} = \{(\text{F3,True})\}$$

  The Possibility Space after applying the refinement is shown in Figure 5.12 (Option 1). The reconfigurations removed are represented with dashed lines.

- If the designer decides to remove $R3$. The refinement modifies the guard of the resolution $R3$ to avoid its execution on configuration s2:

$$\left[!s2\right]\ R3_{cond1} = \{(\text{F4,False})\}$$

  The Possibility Space after applying the refinement is shown in Figure 5.12 (Option 2). Dashed lines mean removed reconfigurations by the refinement.

- If the designer decides to modify the condition that triggers one of the simultaneous resolutions, for example $R3$, the refinement modifies the resolution as follows:

$$R3_{cond3} = \{(\text{F4,False})\}$$

Figure 5.12 (Option 3) shows the Possibility Space after applying the last variant of the refinement.

5.6 Discussion through the application of properties

As we stated in previous chapter (Chapter 4), the design method is an iteratively process. This process intends to improve the design of self-adaptive systems through systematic refinements. Designers are free to select the properties that best suit the requirements of the system. Since the properties are applied successively, the resulting Possibility Space of one property is used as the starting point for the next property.

Designers are not forced to follow preestablished sequences of properties. The order of application of properties only depends on the designer's decisions. Initially, all the properties are independent among them. That is to say, the effect produced by a property neither cause side effects nor depend on other properties. Nevertheless, we plan to carry out a thorough analysis to detect possible dependencies among properties. As future work, we will provide guidance to the designers when selecting properties, in order to guarantee orthogonal properties through the refinement process.

5.7 Conclusions

This chapter identifies potential design properties for self-adaptive systems. Specifically, five valuable properties have been defined: Safe Reconfigurations and Reachability, Redundancy, Reversibility, Contextual Consistency and Contextual Determinism. For each property, we have defined the corresponding refinements to automatically fulfill the properties. The refinements always keep the invariant: improving the design of the system but not a the expense of losing reconfiguration capability. The suggested properties are only an example; designers could include more properties to cover different aspects of systems design and execution. Dealing with design properties optimizes design and, consequently, the system execution.

Chapter 6

Case Study: the Smart Hotel

In this chapter, we put in practice the design method proposed in this master thesis. The method has been applied in a previous case study: the Smart Hotel (Cetina et al., 2010). The Smart Hotel is a self-adaptive system which reconfigures its services autonomically according to changes in the surrounding context.

As starting point, we use a reduced version of the Smart Hotel specification. The original Smart Hotel presents 39 features and 8 resolutions. Since the number of possible configurations grows combinatorially with the number of variation points and variants, for illustration purposes, we use a reduced version with 18 features and 4 resolutions. Following the design method, all the design properties have been successively applied. The case study illustrates the difficulty to specify systems of this kind and the significant enhancement introduced after applying different design properties.

The design approach proposed significantly improves the ability to develop and maintain self-adaptive systems, by providing mechanisms to analyze run-time reconfiguration effects and to automatically refine

specifications ensuring valuable execution issues.

The reminder of this chapter is structured as follows. Section 6.1 introduces the Smart Hotel system. Section 6.2 following the design method proposed in this thesis, all the design properties have been applied in the Smart Hotel specification. Finally, Section 6.3 concludes the chapter.

6.1 Overview of the Smart Hotel Case Study

The Smart Hotel reconfigures its services and devices according to changes in the surrounding context. In the Smart Hotel, different customers use the same room over time. Each client has their own preferences for the room and it should be adjusted to improve the customer's stay. Furthermore, the preferences of a user change depending on the activities performed. For example, preferences change depending on whether the customer is sleeping or working. A hotel room changes its features (activating or deactivating these features) in order to make the stay as pleasant as possible. The Smart Hotel case study was deployed with real devices (EIB-KNX and RFID).

In particular, the Smart Hotel uses variability models at run-time to drive the reconfigurations of the system: how a system can activate or deactivate its own features dynamically at run-time by fulfilling certain context conditions. As starting point, we use a reduced version of the original Smart Hotel specification in order to facilitate understanding of the design method. According to the Feature Modeling technique, the Smart Hotel Feature Model presents 18 features (Figure 6.1). The Feature Model represents the different room configurations. The green boxes represent the smart hotel active features (current features), while the red boxes represent inactive fetures (potential variants that may be activated in the future). The initial configuration of the Smart Hotel consists of the following active features: *SmartRoom*, *Security*, *Alarm*, *InRoomDetection* and *VolumetricSensor*.

The reconfigurations that can be performed by the Smart Hotel are represented by the set of resolutions illustrated in Table 6.1. The

Feature Model and the set of resolutions are used at run-time to drive the Smart Hotel reconfigurations autonomically.

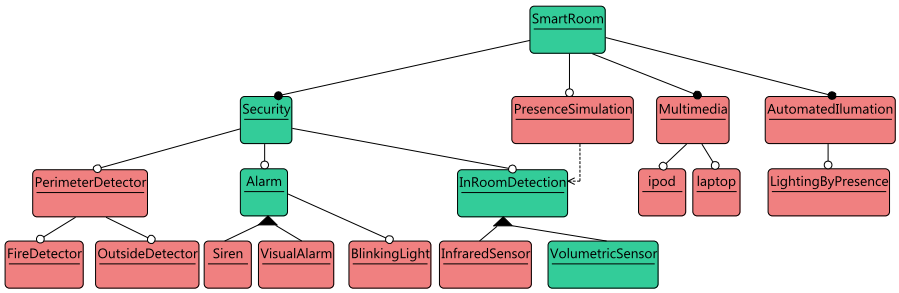


Figure 6.1: Feature Model describing the Smart Hotel case study

RES	COND	DESCRIPTION	FEATURE	STATE
R1	cond1	A person is entering in the room	AutomatedIllumination	True
			InfrarredSensor	False
			LightiningByPresence	True
			PresenceSimulation	False
R2	cond2	A person is leaving the room	AutomatedIllumination	False
			InfrarredSensor	True
			LightiningByPresence	False
			PresenceSimulation	True
R3	cond3	Sleeping	InRoomDetection	False
			VolumetricSensor	False
R4	cond4	Listening to music	ipod	True

Table 6.1: Initial set of resolutions of the Smart Hotel

6.2 Application of the Design Method

This section exemplifies the application of the design method to improve the design of the Smart Hotel. To illustrate the applicability, all the

design properties presented in this thesis have been applied to the Smart Hotel specification.

6.2.1 Obtaining the Possibility Space

Given the Feature Model of the Smart Hotel (Figure 6.1) and the set of resolutions (Table 6.1), it is difficult to foresee the implications that the reconfigurations will have at run-time. By applying the design method proposed, we can automatically obtain a graphical representation of the run-time reconfigurations in a state machine model. This state machine model represents the Possibility Space of the system. From the Smart Hotel specification above, the Possibility Space can be obtained automatically as explained in Chapter 4 (Section 4.2.1). Figure 6.2 illustrates the Possibility Space derived from the Smart Hotel specification.

The definition of 4 resolutions generates a Possibility Space with 12 states and 28 reconfigurations among them. Each state represents a feasible configuration of the system. Each configuration is conformed by a set of active features in the FM. The definition of each configuration is illustrated in Table 6.2. Each column contains a configuration of the Possibility Space. Each row contains a feature of the feature model. Green cells mean, active features in the configuration represented by the column. The table only captures the features modified by the resolutions. The reminder of the features in the FM were inactive initially and will keep inactive during execution.

6.2.2 Reconfiguration Analysis and Refinement

Once the Possibility Space of the Smart Hotel has been obtained, the designer can analyze run-time reconfiguration effects to assure execution issues. Therefore, the designer can select available design properties in the catalog. The next subsections illustrate the application of all the properties to the Smart Hotel specification. Following the design method, the properties are applied iteratively. The resulting Possibility Space of one property is used as the starting point for the next property.

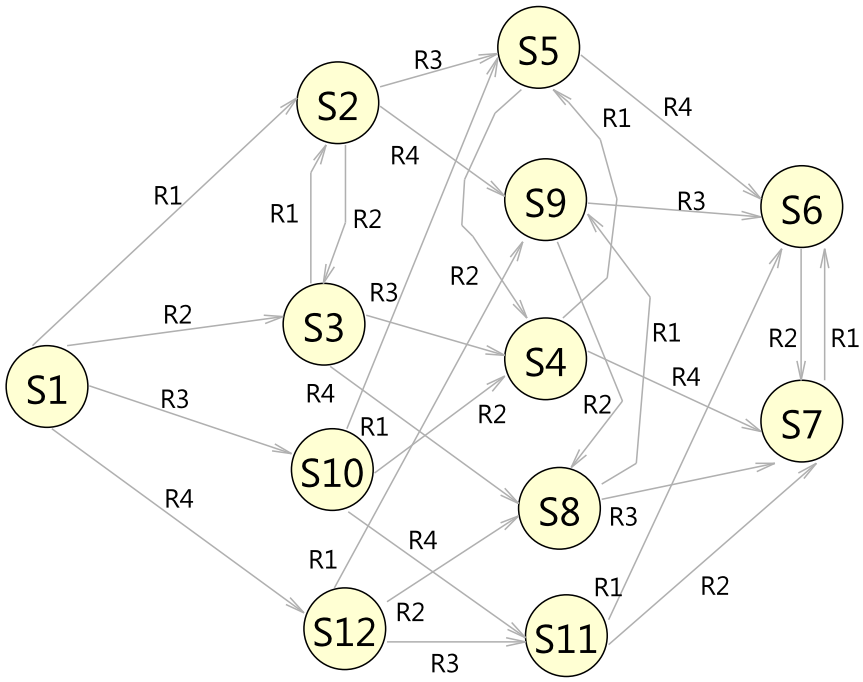


Figure 6.2: Possibility Space of the Smart Hotel

6.2.2.1 Property 1: Safe Reconfigurations and Reachability

The Feature Model of the Smart Hotel contains the following variability constraint:

PresenceSimulation requires InRoomDetection

Given the textual description of the resolutions, it is difficult to foresee if the system will violate the constraint during execution. Therefore, the designer can select the **Safe Reconfigurations and Reachability property** to guarantee that all the reconfigurations are safe. This is, reconfigurations never lead to inconsistent states where the variability

Feature \ State	1	2	3	4	5	6	7	8	9	10	11	12
SmartRoom	■	■	■	■	■	■	■	■	■	■	■	■
Security	■	■	■	■	■	■	■	■	■	■	■	■
Alarm	■	■	■	■	■	■	■	■	■	■	■	■
InRoomDetection	■	■	■						■			■
VolumetricSensor	■	■	■					■	■			■
AutomatedIlumination		■			■	■			■			
InfraredSensor			■	■			■	■				
LightingByPresence		■			■	■			■			
PresenceSimulation			■	■			■	■				
ipod						■	■	■			■	■

Table 6.2: Configurations defined by the Possibility Space

constraint is violated.

Figure 6.3 illustrates the Smart Hotel Possibility Space with invalid configurations and unsafe reconfigurations highlighted in color red. In the Possibility Space there are two invalid configurations: S8 and S12, since in both configurations feature *PresenceSimulation* is active while feature *InRoomDetection* is inactive. Consequently, the reconfigurations that reach S8 or S12 from valid configurations, are unsafe and should be avoided from the specification. The Possibility Space contains 6 unsafe reconfigurations. The unsafe reconfigurations are triggered by resolutions: $R2$ (from configurations S5, S6, S10 and S11) and $R3$ (from configurations S3 and S8).

The refinement associated to this property, automatically avoids these undesirable reconfigurations. The refinement works as follows:

- First, the refinement modifies the guards of resolutions $R2$ and $R3$ to avoid their execution in the configurations that trigger unsafe reconfigurations. The set of resolutions is modified as shown in Table 6.3.

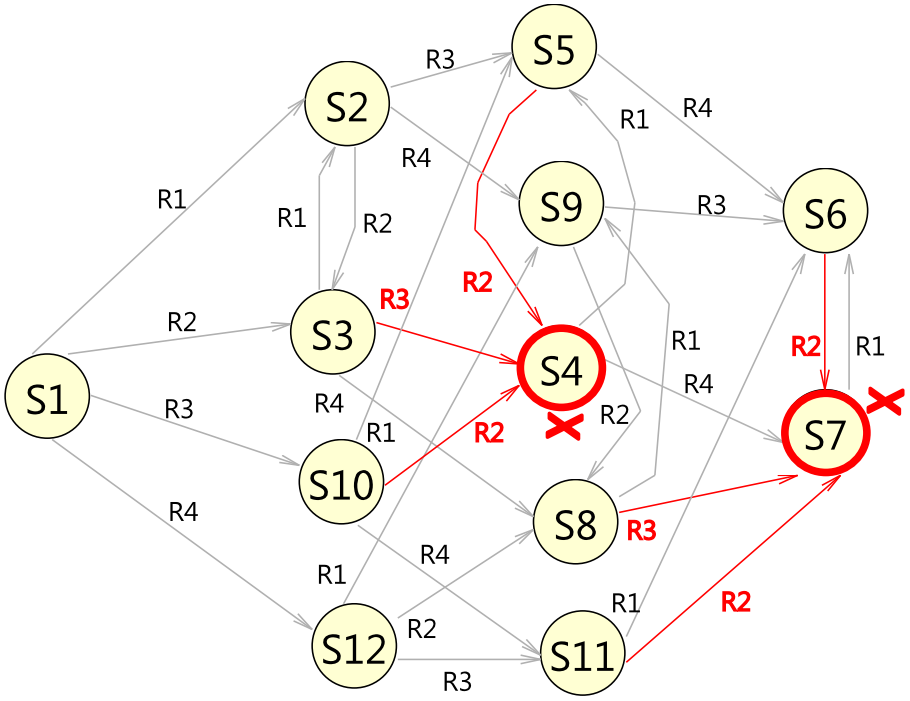


Figure 6.3: Possibility Space with invalid configurations and unsafe reconfigurations identified

- Second, when some reconfigurations are removed other configurations can be unreachable. In order to not loose reconfiguration capability, the refinement calculates new safe reconfigurations which directly lead from the valid source configurations of the old unsafe reconfigurations, to the configurations reachable from the unsafe removed. It also introduces guards to the new resolution in order to avoid the generation of new configurations because of the application of this resolution from other configurations. The refinement generates the new resolutions shown in Table 6.4. These new resolutions are calculated as explained in Chapter 5 (Section 5.1). The Smart Hotel Possibility Space after

RES	COND	GUARD	FEATURE	STATE
R1	cond1		AutomatedIllumination	True
			InfrarredSensor	False
			LightiningByPresence	True
			PresenceSimulation	False
R2	cond2	[!S10; !S5; !S11; !S6]	AutomatedIllumination	False
			InfrarredSensor	True
			LightiningByPresence	False
			PresenceSimulation	True
R3	cond3	$[!S3; !S8]$	InRoomDetection	False
			VolumetricSensor	False
R4	cond4		ipod	True

Table 6.3: Resolutions modified to avoid unsafe reconfigurations

applying this refinement is shown in Figure 6.4. The new reconfigurations generated are represented with thick lines.

RES	COND	GUARD	FEATURE	STATE
$R3{+}R1$	$cond3 \wedge$ $cond1$	$[S3; S8]$	InRoomDetection	False
			VolumetricSensor	False
			AutomatedIllumination	True
			InfraredSensor	False
			LightingByPresence	True
			PresenceSimulation	False
$R3{+}R4{+}R1$	$cond3 \wedge$ $cond4 \wedge$ $cond1$	$[S3]$	InRoomDetection	False
			VolumetricSensor	False
			iPod	True
			AutomatedIllumination	True
			InfraredSensor	False
			LightingByPresence	True
			PresenceSimulation	False
$R2{+}R1$	$cond2 \wedge$ $cond1$	$[S10; S11]$	AutomatedIllumination	False
			InfraredSensor	True
			LightingByPresence	False
			PresenceSimulation	True
			AutomatedIllumination	True
			InfraredSensor	False
			LightingByPresence	True
			PresenceSimulation	False
$R2{+}R4{+}R1$	$cond2 \wedge$ $cond4 \wedge$ $cond1$	$[S10]$	ipod	True
			AutomatedIllumination	False
			InfraredSensor	True
			LightingByPresence	False
			PresenceSimulation	True
			AutomatedIllumination	True
			InfraredSensor	False
			LightingByPresence	True
			PresenceSimulation	False

Table 6.4: New resolutions generated by the refinement

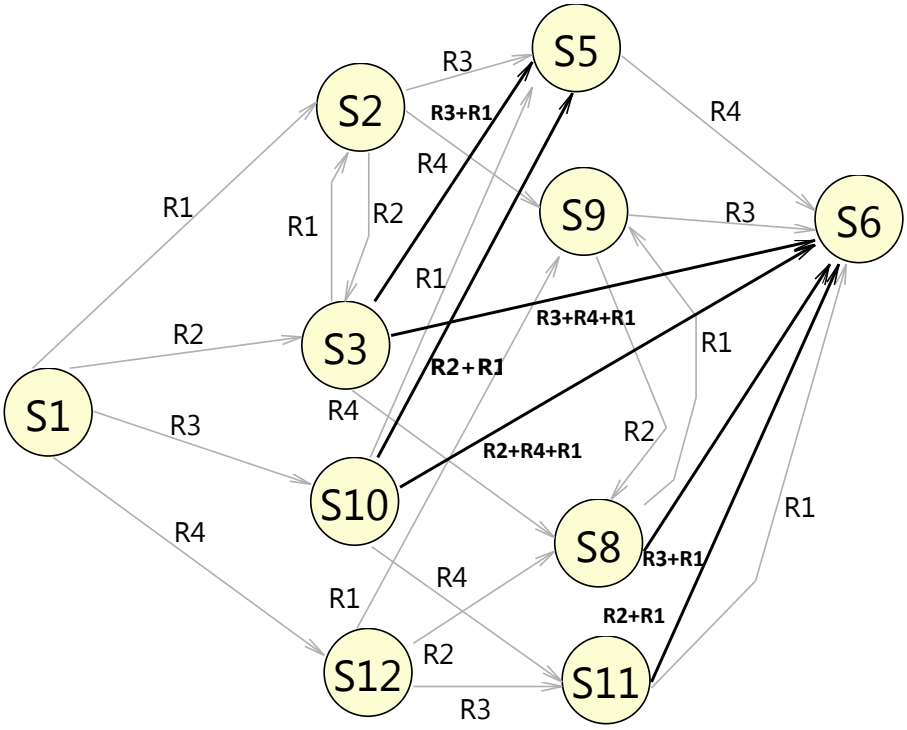


Figure 6.4: Possibility Space after applying Safe Reconfigurations and Reachability refinement

6.2.2.2 Property 2: Redundancy

Eventually, designers can define duplicated behavior unintentionally. Duplicated behavior refers to resolutions that evolve the system between the same source configuration and the same target configuration by means of different actions. Other refinements, can also generate new resolutions that are redundant with existing resolutions in the specification. The **Redundancy property** guarantees that the specification does not contain duplicated behavior.

After applying the Safe Reconfigurations and Reachability refine-

ment, the designer could select the Redundancy property. It can be observed that the resulting Possibility Space (Figure 6.4) of previous refinement contains redundant behavior. The reconfigurations triggered by resolutions R1 and R2+R1 are redundant from configurations S10 and S11, because both evolve the system between the same configurations: S10-S5 and S11-S6. Table 6.5 summarizes the redundant reconfigurations.

RES	SOURCE	DEST	COND	GUARD
$R1$	S10	S5	$cond1$	
$R2{+}R1$	S10	S5	$cond2 \wedge cond1$	$[S10; S11]$
$R1$	S11	S6	$cond1$	
$R2{+}R1$	S11	S6	$cond2 \wedge cond1$	$[S10; S11]$

Table 6.5: Redundant reconfigurations in the Possibility Space

By applying the **Redundancy Avoidance Refinement** to the previous refined Possibility Space (Figure 6.4), the redundant reconfigurations can be avoided automatically . The refinement selects the simplest resolution and removes the others. R1 implies 4 actions whereas R2+R1 implies 8 actions. Therefore, the refinement eliminates R2+R1. The refinement modifies the guard of resolution R2+R1 in order to avoid its execution in the configurations that trigger redundant resolutions (S10 and S11). The resolution R2+R1 has a previous guard $[S10; S11]$, then it only executes in configurations S10 and S11. Since, this refinement avoids the execution of R2+R1 in configurations S10 and S11, this resolution will never execute. Furthermore, in order to not loose reconfiguration capability, the refinement modifies the condition of R1 to include the possibilities of reconfiguration of the removed resolution (R2+R1). Table 6.6 shows the resolutions modified by the refinement.
The Possibility Space after applying the Redundancy Avoidance refinement is shown in Figure 6.5. The reconfigurations removed are represented with red dashed lines.

RES	COND	GUARD
$R2{+}R1$	$cond2 \wedge cond1$	$[S10; S11]$ **[!S10;!S11]**
$R1$	**(cond1) or (cond2∧cond1)**	

Table 6.6: Resolutions modified by the refinement to avoid redundancy

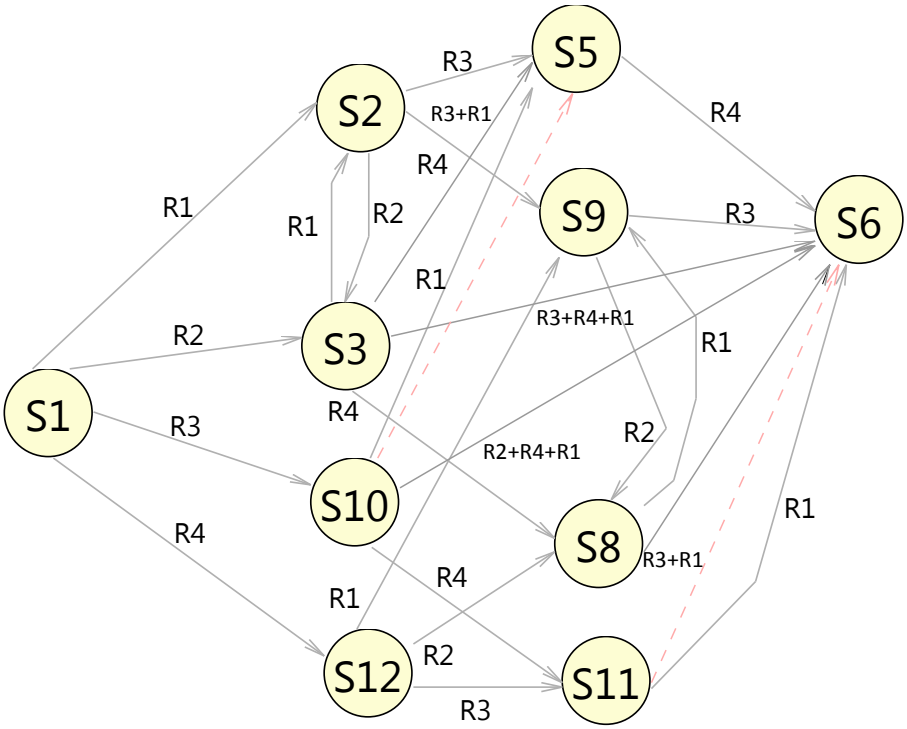


Figure 6.5: Possibility Space after applying the Redundancy Avoidance Refinement.

6.2.2.3 Property 3: Reversibility

The designer can take into consideration rollback capabilities to reconfigurations. Thus, the **Reversibility property** can be used. Given the Possibility Space of Figure 6.5, it can be observed that the space

contains reversible reconfigurations between configurations S2-S3 and S9-S8. These reconfigurations are triggered by resolutions R2 and R1. The reminder of reconfigurations do not have another reconfiguration that leads directly to the source configuration.

Depending on the application domain, a different type of reversible system could be required. Designers can apply one of the following refinements making either a Total Reversible System or a Nonreversible System:

- **Nonreversible System Refinement**

The purpose of this refinement is to ensure that the effects of each reconfiguration cannot be undone by means of other reconfiguration at runtime. In the Smart Hotel Possibility Space of Figure 6.5, there are two reversible resolutions, $R1$ and $R2$, between configurations: $S2$-$S3$ and $S8$-$S9$. Nevertheless, other configurations ($S1$, $S10$, $S12$ and $S11$) can evolve by means of $R1$ or $R2$ to different configurations, but there are no reversible reconfigurations to them. Designers can apply the **Nonreversible System Refinement** to automatically achieve specifications free of reversible reconfigurations. The refinement adds guard to the resolution R1 or R2 to avoid their execution in the configurations that trigger reversible reconfigurations, but keeping them in the reminder of configurations. In this way, reconfiguration capability is not lost. By observing the textual description of the resolutions, it is difficult to foresee the configurations where $R1$ and $R2$ will be reversible, and the configurations where there will not be a reversible reconfiguration after triggering $R1$ or $R2$. Thus, the design method can substantially help designers by analyzing and automatically refining the specification to fulfill this requirement. The designer can choose which of these two resolutions should be removed. Consequently, two different Possibility Spaces can be obtained:

- If the designer decides to remove reconfiguration R1, the refinement modifies the guard of the resolution R1 to avoid its execution on configuration $S3$ and $S8$. Figure 6.6 shows the resulting Possibility Space after applying the refinement. The reconfigurations removed are represented with red dashed lines.

RES	COND	GUARD
$R1$	(cond1) or (cond2∧cond1)	$[!S3; !S8]$

Table 6.7: Resolution R1 after applying the Nonreversible System refinement removing R1

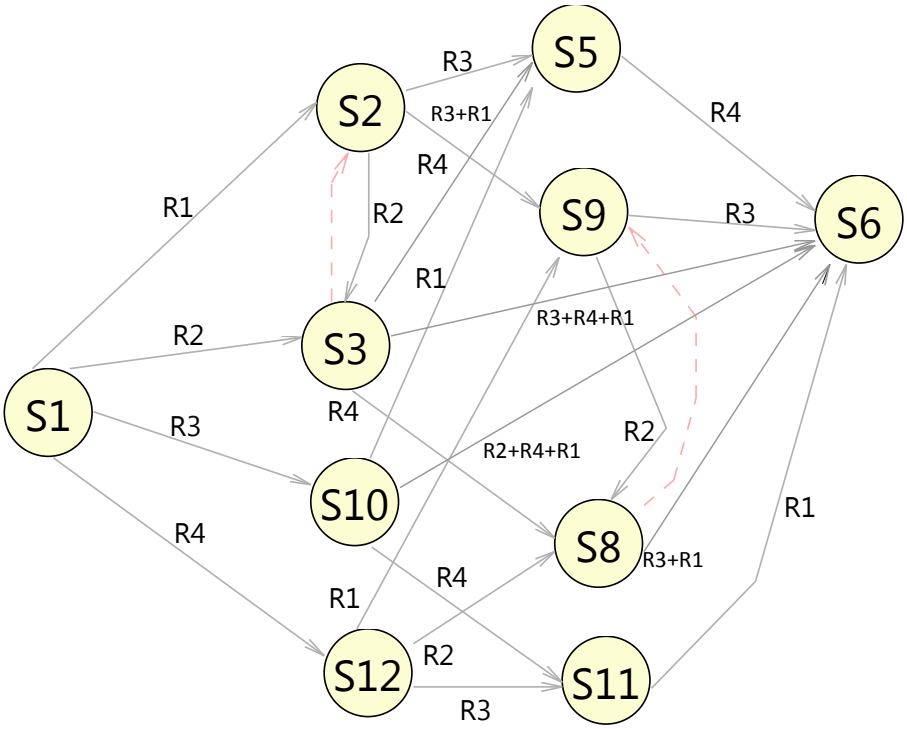


Figure 6.6: Possibility Space after applying the "Nonreversible System" refinement removing $R1$

- If the designer decides to remove reconfiguration R2, the refinement modifies the guard of the resolution R2 to avoid its execution on configuration S2 or S9. Figure 6.7 shows the resulting Possibility Space after applying the refinement. Red dashed lines mean reconfigurations removed.

RES	COND	GUARD
$R2$	$cond2$	$[!S10; !S5; !S11; !S6]$ $[!S2; !S9]$

Table 6.8: Resolution R2 after applying the Nonreversible System refinement removing R2

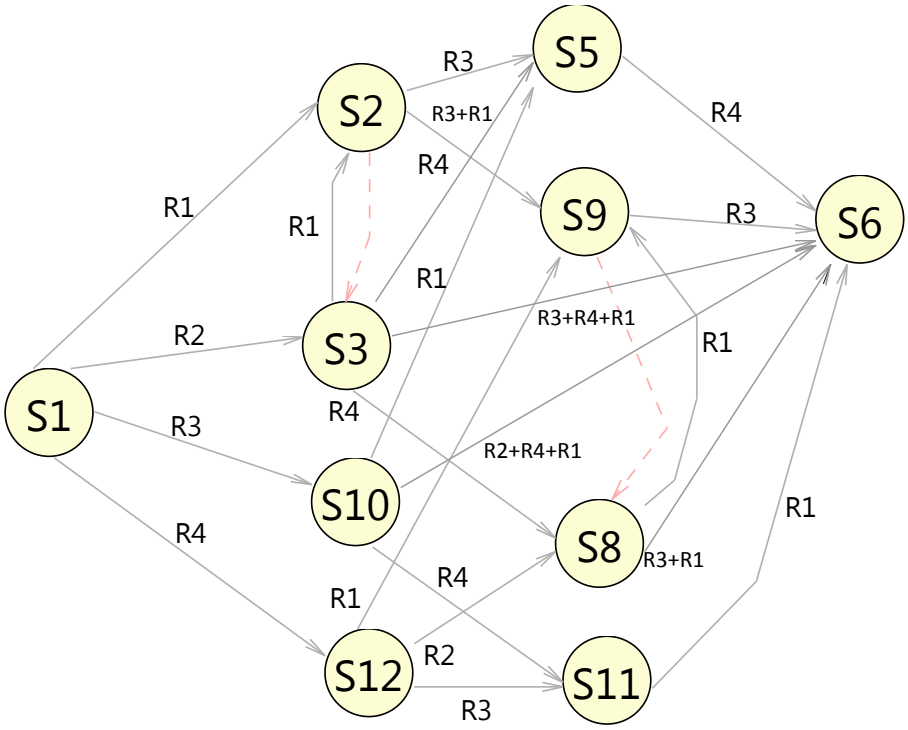


Figure 6.7: Possibility Space after applying the "Nonreversible System" refinement removing $R2$

- **Total Reversible System Refinement**

If the designer is interested in guaranteeing that, for all reconfigurations occurring in the system, it is always possible to undo the effect of the reconfiguration, the **Total Reversible System refinement** can be

applied. To fulfill this requirement, the refinement generates six new resolutions illustrated in Table 6.9.

RES	GUARD	FEATURE	STATE
$\overline{R3}$	$[S5; S11; S6]$	InRoomDetection	True
		VolumetricSensor	True
$\overline{R4}$	$[S12; S9; S8; S11; S6]$	iPod	False
$\overline{R3}$+$\overline{R1}$	$[S5; S6]$	InRoomDetection	True
		VolumetricSensor	True
		AutomatedIlumination	False
		InfraredSensor	True
		LightingByPresence	False
		PresenceSimulation	True
$\overline{R3}$+$\overline{R4}$+$\overline{R1}$	$[S6]$	iPod	False
		AutomatedIlumination	True
		InfraredSensor	False
		LightingByPresence	True
		PresenceSimulation	False
		AutomatedIlumination	False
		InfraredSensor	True
		LightingByPresence	False
		PresenceSimulation	True
$\overline{R1}$	$[S2; S5; S6]$	AutomatedIlumination	False
		LightingByPresence	False
$\overline{R2}$	$[S3; S8]$	InfraredSensor	False
		PresenceSimulation	False

Table 6.9: New resolutions generated by the Total Reversible refinement

The refinement also introduces guards to the new resolutions in order to avoid the generation of new configurations because of the application of this resolution from other configurations. Figure 6.8 shows the Possibility Space after applying the refinement. The generated resolutions are represented with thick lines. As the figure shows, a great number of reconfigurations are needed to fulfill this requirement. By

visual inspection, it would be very difficult to foresee and to specify the required resolutions.

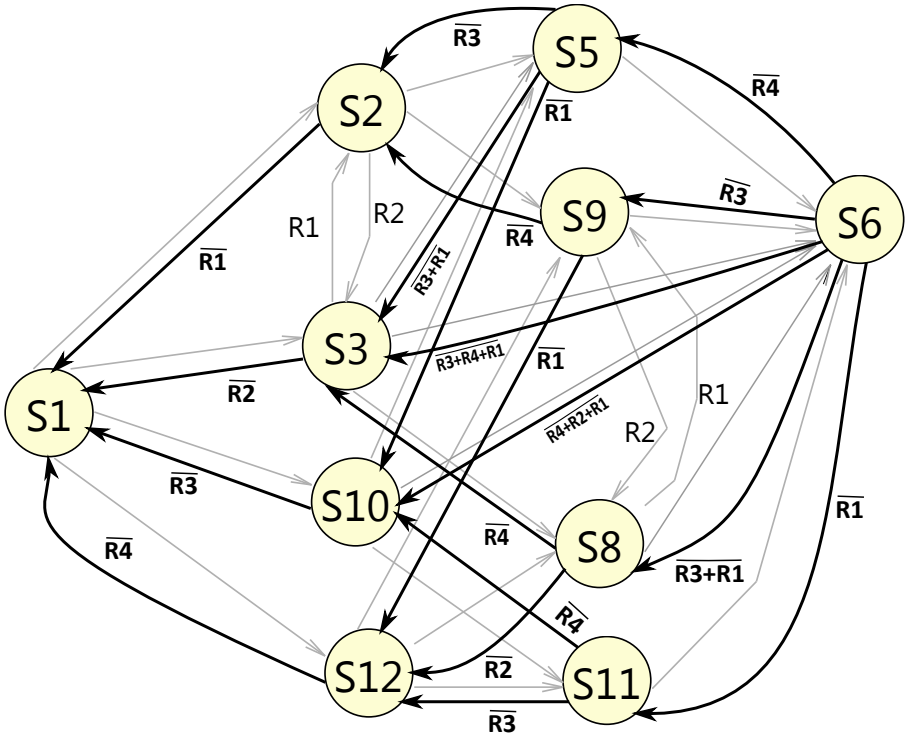


Figure 6.8: Possibility Space after applying the "Total Reversible System" refinement

6.2.2.4 Property 4: Contextual Consistency

This property guarantees that when a determined context condition is fulfilled, the system evolves (independently from its current state) to a predefined or controlled configuration. For example, if in the Smart Hotel is detected a fire, the system must always reconfigure to a configuration *"Emergency"* independently of the current system configuration. In the new configuration "Emergency" are active the following features: *SmartRoom, Security, Alarm, Siren and BlinkingLight.*

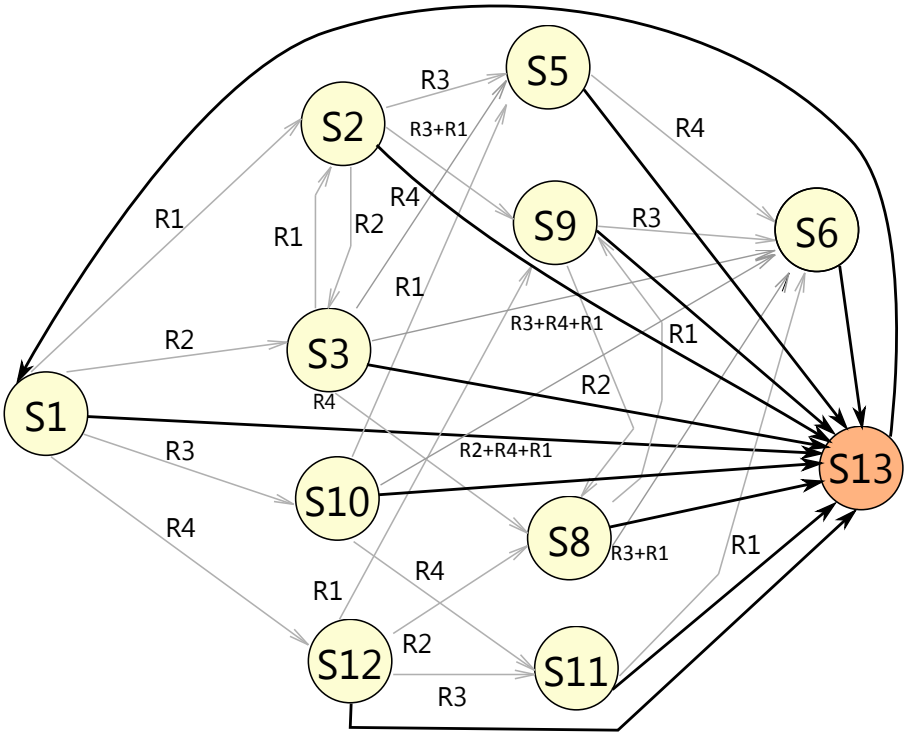


Figure 6.9: Possibility Space after applying the "Contextual Consistency" refinement

The **Contextual Consistency refinement** generates from each configuration in the Possibility Space a new reconfiguration that evolves the system to the new configuration "Emergency". Figure 6.9 illustrates the Possibility Space after applying this refinement. The configuration "Emergency" is represented by the state S13. The new generated reconfigurations are represented with thick lines. The refinement generates twelve new resolutions presented in Table 6.10.

RES	COND	GUARD	FEATURE	STATE
R_{S1-S13}	cond_{fire}	$[S1]$	InRoomDetection	False
			VolumetricSensor	False
			Siren	True
			BlinkingLight	True
R_{S2-S13}	cond_{fire}	$[S2]$	InRoomDetection	False
			VolumetricSensor	False
			AutomatedIllumination	False
			LightingByPresence	False
			Siren	True
			BlinkingLight	True
R_{S3-S13}	cond_{fire}	$[S3]$	InRoomDetection	False
			VolumetricSensor	False
			InfraredSensor	False
			PresenceSimulation	False
			Siren	True
			BlinkingLight	True
R_{S5-S13}	cond_{fire}	$[S5]$	AutomatedIllumination	False
			LightingByPresence	False
			Siren	True
			BlinkingLight	True
R_{S6-S13}	cond_{fire}	$[S6]$	AutomatedIllumination	False
			LightingByPresence	False
			iPod	False
			Siren	True
			BlinkingLight	True

R_{S8-S13}	cond_{fire}	$[S8]$	InRoomDetection	False
			VolumetricSensor	False
			InfraredSensor	False
			PresenceSimulation	False
			iPod	False
			Siren	True
			BlinkingLight	True
R_{S9-S13}	cond_{fire}	$[S9]$	InRoomDetection	False
			VolumetricSensor	False
			AutomatedIlumination	False
			LightingByPresence	False
			iPod	False
			Siren	True
			BlinkingLight	True
$R_{S10-S13}$	cond_{fire}	$[S10]$	Siren	True
			BlinkingLight	True
$R_{S11-S13}$	cond_{fire}	$[S11]$	iPod	False
			Siren	True
			BlinkingLight	True
$R_{S12-S13}$	cond_{fire}	$[S12]$	InRoomDetection	False
			VolumetricSensor	False
			iPod	False
			Siren	True
			BlinkingLight	True
R_{S13-S1}	$\text{cond}_{notfire}$	$[S13]$	Alarm	False
			Siren	False
			BlinkingLight	False
			InRoomDetection	True
			VolumetricSensor	True

Table 6.10: Generated Resolutions after applying Contextual Consistency refinement

6.2.2.5 Property 5: Contextual Determinism

This property involves predictability in the reconfiguration process to future states. A system is deterministic if for each possible context condition, every state in the system has exactly one reconfiguration that leads to the next state. Thus, when a determined context condition is fulfilled, the system can only reconfigure to one destination state. On the contrary, if more than one condition is fulfilled at the same time, several reconfigurations will execute at the same time, and the system is indeterministic. Indeterministic behavior is unacceptable in the Smart Hotel, since two different room configurations at the same time has no sense.

The Smart Hotel specification defines a deterministic system because each resolution is associated to a different context condition. Let's suppose that now the hotel room is equipped with a new configuration suitable for working. Then the designer defines a new resolution $R5$, as illustrated in Table 6.11.

RES	COND	DESCRIPTION	FEATURES	STATE
$R4$	*cond4*	A person starts listening to music	ipod	True
R5	**cond4**	**A person starts working**	**laptop**	**True**

Table 6.11: New defined resolution

Unintentionally, the designer has set the condition that trigger $R5$ equal to the condition that trigger an existing resolution ($R4$). The definition of the resolution $R5$ generates a nondeterministic Smart Hotel. During execution, the fulfillment of the context condition *cond4* will lead to more than one reconfiguration (triggered by R4 and R5), therefore, to more than one destination state. Designers can apply the **Deterministic System refinement** to guarantee that the Smart Hotel is deterministic at run-time.

The refinement offers two options:

- Allows designers to choose the resolution that should be triggered when the context condition is fulfilled.

- Modifies the condition of some resolutions in order to avoid the simultaneous reconfigurations.

6.3 Conclusions

This chapter has illustrated the application of the design method proposed in this work to refine the specification of an existing self-adaptive system: the Smart Hotel. The above case study has demonstrated the difficulty to foresee and control the behavior of self-adaptive systems through variability specifications.

Reconfigurations at run-time can have a great number of repercussions and situations. Thus, techniques to analyze the effects of run-time reconfigurations can improve the system execution and make users feel more comfortable with self-adaptive systems. Furthermore, the explosion of states makes the design of self-adaptive systems a tedious task that is prone to errors. The use of guidelines to specify self-adaptive systems and analyze reconfigurations helps engineers to develop adaptive systems of higher quality. All the design properties identified in this master thesis haven been successively applied to the case study and the benefits are immediate.

Chapter 7

Tool Support

The design method proposed in this master thesis follows a model-based approach for the development of self-adaptive systems. A model is a simplification of a system, built with an intended goal in mind, that should be able to answer questions in place of the actual system (Bézivin & Gerbé, 2001). In this work, a series of models are used to describe a self-adaptive system and drive its behavior at run-time.

MDE proposes the use of metamodels to formalize concepts and their relationships. A metamodel defines the constructs that can be used to describe systems. Metamodels are useful to formalize and exchange models. We have defined several metamodels that formalize the concepts used in our design method to model self-adaptive systems.

The modeling community has developed several projects to support the MDE paradigm under the Eclipse Modeling Project. Different tasks comprised by the MDE approach are supported: definition of a modeling language (metamodeling), description of a system using this language (modeling). Eclipse Modeling Framework (EMF)[1] provides tool support

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

for the definition of metamodels and the edition of models. For the implementation of the graphical tools we have used the possibilities offered by the Eclipse Graphical Modeling Framework (GMF) which is part of the Eclipse Modeling Project. GMF provides a generative component and run-time infrastructure for developing graphical editors based on EMF.

This chapter formalizes the concepts used for describing self-adaptive systems through metamodels. We have implemented some graphical editor tools based on Eclipse in order to facilitate the specification of this kind of systems. Eclipse tools are defined by combining a set of plug-ins with different functionalities. We have developed some plugins to support the technique presented for the design of self-adaptive systems. Furthermore, we have integrated existing plug-ins that provide feature modeling capabilities that meet our requirements.

The following sections (7.1, 7.2, 7.3, 7.4) provide detail on the definition of the tools and the graphical environments to design self-adaptive systems. Section 7.5 explains how the design method is automated by using the tools defined and model transformations. Finally, Section 7.6 concludes the chapter.

7.1 The Feature Model Metamodel

In order to model the variability of a self-adaptive system we have used Moskitt Feature Modeler (MFM). MFM is a free open-source tool that is part of the Moskitt modeling suite[2]. MFM is defined as a set of plug-ins that we could incorporate to enhance our tool support with feature modeling capabilities. Figure 7.1 shows the different concepts in the Feature Model metamodel and the relationships among them. The metaclass *FeatureModel* is used as the root element of Feature models. The *Feature* metaelement represents the different features of the Feature model. Each feature can have attributes that are represented by the *Attribute* metaelement. Features are related among them through relationships represented by the *CardinalityBasedRelationship* metaele-

[2]http://www.moskitt.org

ment. This metaelement is specialized in the different relationships that the Feature model supports: *Or, Alternative, Optional, and Mandatory.*

Figure 7.2 shows the environment of the MFM.

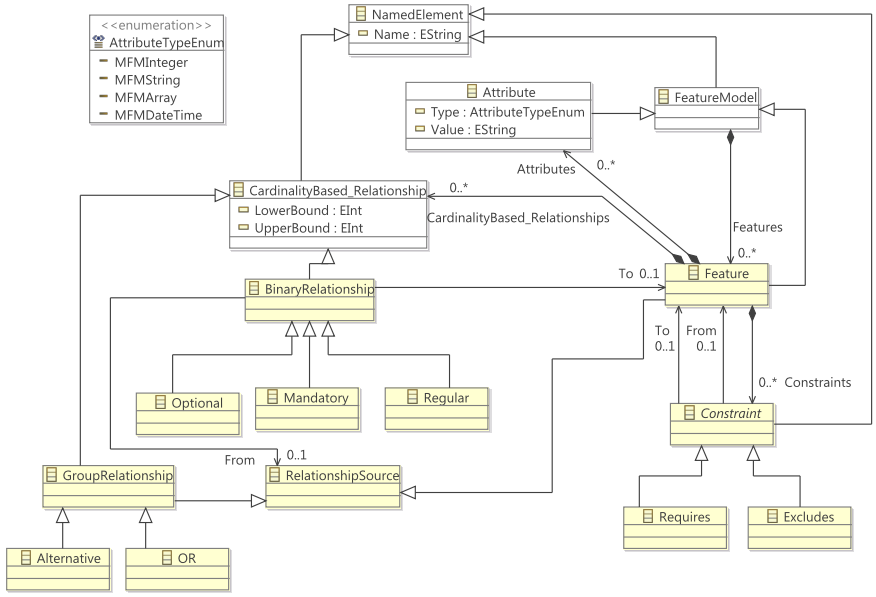


Figure 7.1: The Feature Model metamodel

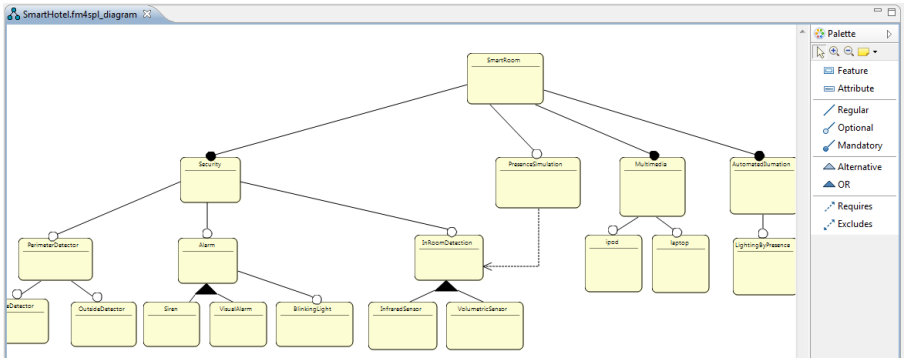


Figure 7.2: MFM environment

7.2 The Configuration Metamodel

The Configuration metamodel defines the concepts to represent feasible configurations of a Feature Model. The metamodel is shown in Figure 7.3.

The *ConfigurationModel* is the root element of the metamodel. The ConfigurationModel has a *FeatureModel* from the Feature Model metamodel, and is composed by a set of *FeatureState* metaelements. Each FeatureState has a *Feature* from the *FeatureModel* and represents the state of that feature. There are three types of states: *Active, Deactive or Discarded*, meaning that the feature is active, inactive o it does not matter its state in that configuration respectively.

A graphical editor is provided to describe possible configurations of a feature model. Figure 7.4 illustrates the environment of the editor. Features are represented with different colors depending on their state: *green* features represent active features, *red* features represent deactive features and *orange* features represent discarded features. With this tool developers can determine the initial configuration of the system in an intuitive manner.

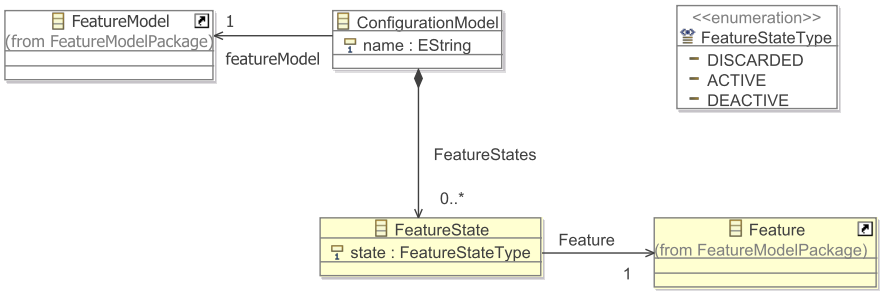


Figure 7.3: The Configuration metamodel

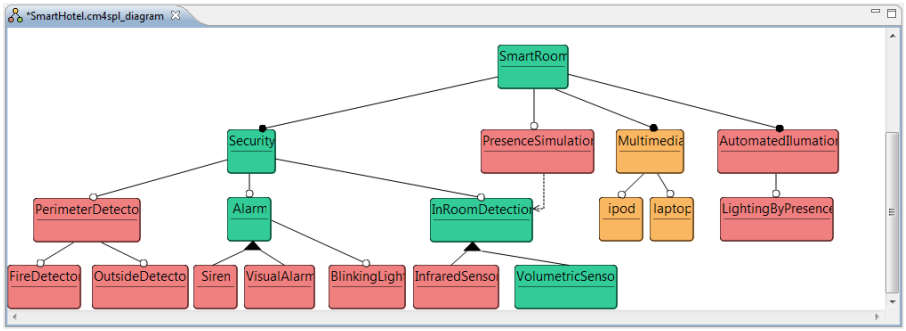


Figure 7.4: Graphical editor for Configuration Models

7.3 The Resolution Metamodel

The Resolution metamodel (see Figure 7.5) defines the concepts used for describing resolutions. The metaclass *ResolutionModel* is the root element of the metamodel. The *ResolutionModel* is composed by a set of *Resolution* metaelements that represent different resolutions. Each *Resolution* is composed by a *Condition* and a set of *Actions*. Furthermore, a *Resolution* can have a *Guard*.

An editor is provided to support the definition of the resolution

model using the EMF capabilities. Figure 7.6 shows the environment of the editor.

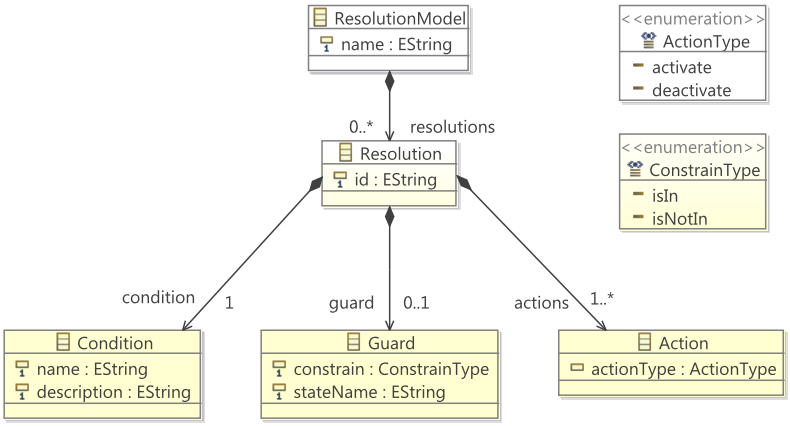


Figure 7.5: The Resolution metamodel

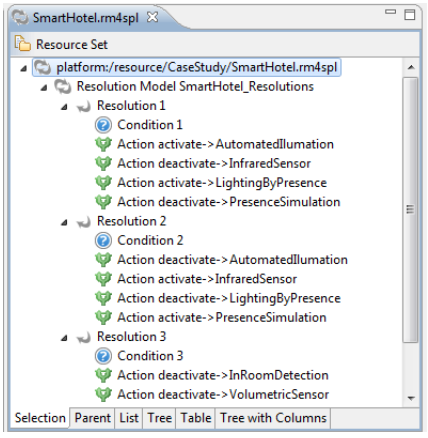


Figure 7.6: Graphical editor for Resolution models

7.4 The State Machine Metamodel

The State Machine metamodel defines the concepts for describing the Possibility Space of the system. The metamodel is shown in Figure 7.7. The *StateMachineModel* is the root element of the metamodel. A *StateMachineModel* is composed by a set of *State* metaelements. Each State has a set of *Transitions*, and each *Transition* has a source state and a destination state.

Figure 7.8 shows the graphical editor for the state machine model.

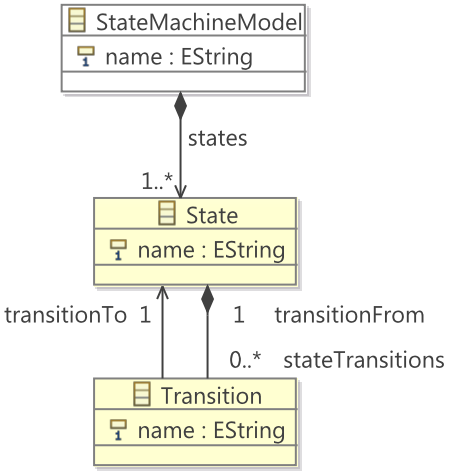


Figure 7.7: The State Machine metamodel

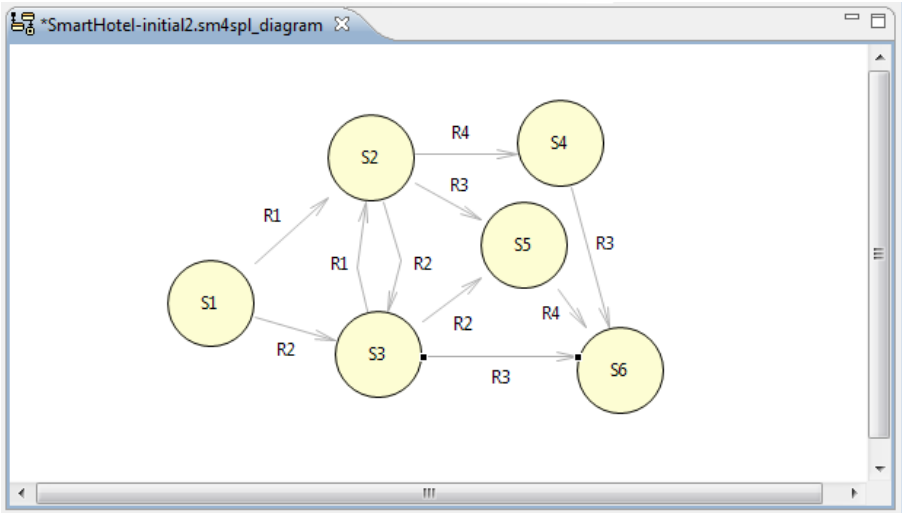


Figure 7.8: Graphical editor for State Machine models

7.5 Automating the Design Process

One of the main reasons for following a Model Driven Engineering (MDE) development is that it is focused on automation. Models can be transformed automatically into new models or code by means of model transformation techniques. This enables automation in system development since software artifacts can be derived in a systematic way. Many technologies and standards give support to this development paradigm. The Object Management Group (OMG) defined Model Driven Architecture (MDA)(Miller & Mukerji, 2003) to provide support to these ideas with standards for metamodeling and the definition of model transformations. Either following MDA or any other paradigm based on MDE ideas, software development can be improved by the raise in the abstraction level that the use of models provides.

The model-driven process defined in this work begins with the specification of a self-adaptive system by means of: a *Feature Model*, an

initial *Configuration Model* and a *Resolution Model*. Using the tools presented above, designers can define the initial system specification. Then, this system specification is automatically transformed into a State Machine model that represents the Possibility Space of the system. Designers can modify the obtained model to guarantee design properties that fulfill assertions about behavior at run-time.

In order to automate the design method, we have implemented a series of model-to-model transformations. Different model-to-model transformation languages exist, such as QVT (OMG, 2005), ATL (Jouault & Kurtev, 2006) or RubyTL (Cuadrado et al., 2006). For these transformations, Atlas Transformation Language (ATL) was used. ATL is a declarative language with Eclipse-based tools support for the edition and execution of model transformations.

Figure 7.9 illustrates an overall view of the transformation process, where are shown the input and output models of each transformation and the metamodels to which they are conformed.

Using ATL, first we have implemented the transformation T0. This transformation defines the mappings, by means of model-to-model rules, to generate a State Machine Model (defined according to the State Machine metamodel). This transformation takes as input three models: *Feature Model*, *Configuration Model* and *Resolution Model*; and generates a State Machine Model. The mapping is unidirectional.

Once the State Machine model has been obtained, several transformations can be applied successively to guarantee different design properties. For each of the properties presented in the catalog (Section 5), the associated refinements have been implemented as model-to-model transformations. All the transformations take as input two models: the *State Machine Model* obtained in the previous transformation, and the *Resolution Model*; and generate a *Refined State Machine Model* and a *Refined Resolution Model*. We denote this transformations T-REF. There are six different transformations T-REF (T-safe, T-redun, T-rev, T-non-rev, T-cons and T-det) that implement the different refinements proposed. At the moment of writing this document, we have already implemented transformations T0, T-redun, T-rev, T-non-rev, T-cons

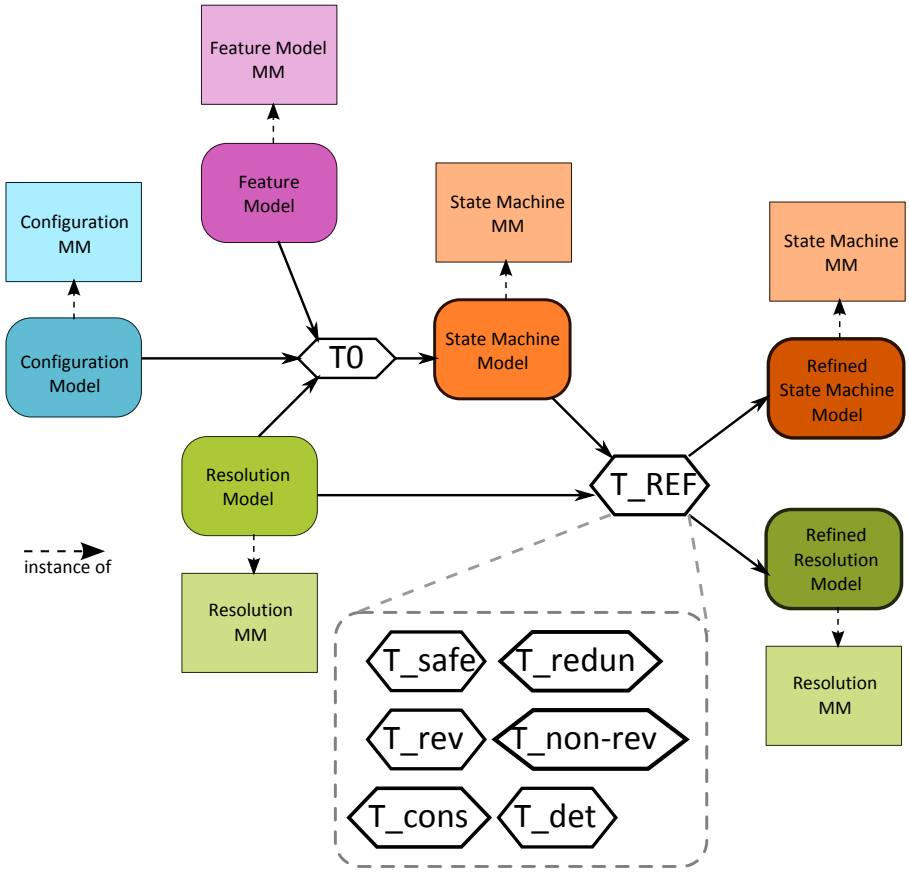


Figure 7.9: Overall view of the transformation process

and T-det, and we are working on the implementation of the T-safe refinement.

7.6 Conclusions

This chapter introduces the set of tools that support the design method proposed in this master thesis. We have used the Eclipse Modeling technologies. By applying MDE principles, we have formalized the concepts involved in the modeling of self-adaptive systems and we have defined a series of editors to ease their specification. The design method for self-adaptive systems is automated by a series of model-to-model transformations. The transformations have been implemented using the ATL transformation language.

Chapter 8

Conclusions

The present work has introduced a model-driven development method for designing self-adaptive systems. This chapter reviews our central results and primary contributions, and proposes new areas for future research in connection with the limitations of this work.

First, Section 8.1 presents the main contributions to the self-adaptive systems community. Section 8.2 provides some information about the conferences where the work has been submitted. Finally, Section 8.3 outlines the ongoing and future work.

8.1 Contributions

The main contribution of this work is a model-based method for the systematic design of trustworthy self-adaptive systems. The work provides the following contributions:

Systematic design approach. Adaptability poses new challenges to Software Engineering. A model-based development method

has been defined to guide developers in the design of self-adaptive systems. The purpose of this method is (1) to analyze the run-time reconfiguration effects at design time and (2) to automatically refine reconfigurations to ensure valuable design issues.

Properties catalog. The explosion of configurations and reconfigurations is one of the major challenges that self-adaptive system engineers have to face. The implications of run-time reconfigurations are very difficult to foresee and control at design-time. Thus, potential design issues and guidelines to assist engineers in the design of self-adaptive systems have been highlighted. Five interesting properties have been identified: Safe Reconfigurations and Reachability, Redundancy, Reversibility, Contextual Consistency and Contextual determinism. These properties can be automatically ensured through refinements. Dealing with design properties optimizes self-adaptive design and, consequently, the execution. The suggested properties are only an example; designers could include more properties to cover different aspects of systems design and execution.

Tool support. We have provided a set of tools that support the design method proposed in this master thesis. These tools ease the modeling of self-adaptive systems, and automate the analysis and refinement of system specifications.

Moreover, in order to evaluate the approach, we have applied the design method to improve the design of a Smart Hotel self-adaptive system. The case study has demonstrated the difficulty to foresee the behavior of self-adaptive systems through variability specifications. This design approach significantly improves the ability to develop and maintain self-adaptive systems, by providing mechanisms to analyze run-time reconfiguration effects and automatically refine specifications to ensure valuable design issues. It is remarkable, that the method improves the design but not a the expense of losing reconfiguration capability.

8.2 Publications

Part of the work proposed in this master thesis was submitted to the following international conferences:

- **SPLC 2011: 15th International Software Product Line Conference**. Although the review comments were very positive, the paper was not selected for presentation at the conference. The supervisors highlighted the originality and the relevance of the research topic for the community. Therefore, we realize that the work developed is a very promising research line.

- **ASE 2011: 26th IEEE/ACM International Conference On Automated Software Engineering** (Tier-A according to CORE conference ranking). The work improved with the feedback provided by SPLC supervisors, was submitted to ASE conference. ASE is one of the most prestigious conferences in the Software Engineering community. We are currently waiting for the acceptance notification next 18th July 2011.

8.3 Future Work

The research presented here is not a closed work and there are several interesting future directions. The following summarizes the research activities that are planned to continue this work.

Improving adaptation visualization. At the moment, our tool supports the visualization of the Possibility Space of the system. We are aware of scalability problems inherent to approaches that have to cope with combinatorial explosion of variants. Thus, we are planning to improve the visualization of Possibility Spaces in order to make feasible the management of large-scale models.

Providing a conceptual framework. We intend to provide a conceptual framework to enable engineers to incorporate more design

properties as required. Furthermore, we plan to study dependencies between design properties in order to guarantee orthogonal properties.

Enabling End-user participation in the Design of self-adaptive systems. Run-time reconfigurations may imply assumptions about the desirable functionality of end-users. We plan to involve end-users in the design of reconfigurations in order to minimize the mismatch between user expectations and system behavior. Specifically, we intend to enable end-users and technical designers participate cooperatively on a design method for self-adaptive systems.

Addressing other Application Domains. We also expect to apply the design approach to develop systems in different application areas, such as Service-Oriented Architecture (SOA) or Method Engineering (ME).

ACKNOWLEDGEMENT

We would like to gratefully thank Dr. Carlos Cetina for his contribution to the initial ideas of this master thesis.

Bibliography

Ahead (2009). Ahead tool suite. http://www.cs.utexas.edu/users/schwartz/ats.html;. accessed november 2009.

Bellotti, V., & Edwards, K. (2001). Intelligibility and accountability: human considerations in context-aware systems. *Hum.-Comput. Interact.*, *16*, 193–212.

Bencomo, N., Blair, G., & France, R. (2008a). Model-driven software adaptation: report on the workshop m-adapt at ecoop 2007. In *Proceedings of the 2007 conference on Object-oriented technology*, ECOOP'07, (pp. 132–141). Berlin, Heidelberg: Springer-Verlag.

Bencomo, N., Blair, G., & France, R. (2008b). Model-driven software adaptation: report on the workshop m-adapt at ecoop 2007. In *Proceedings of the 2007 conference on Object-oriented technology*, ECOOP'07, (pp. 132–141). Berlin, Heidelberg: Springer-Verlag.

Bencomo, N., Grace, P., Flores, C., Hughes, D., & Blair, G. (2008c). Genie: supporting the model driven development of reflective, component-based adaptive systems. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, (pp. 811–814). New York, NY, USA: ACM.

Bézivin, J., & Gerbé, O. (2001). Towards a precise definition of the OMG/MDA framework. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, (p. 273). Washington, DC, USA: IEEE Computer Society.

Blair, G., Bencomo, N., & France, R. B. (2009). Models@ run.time. *Computer*, *42*, 22–27.

Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., & Shaw, M. (2009). Software engineering for self-adaptive systems. chap. Engineering Self-Adaptive Systems through Feedback Loops, (pp. 48–70). Berlin, Heidelberg: Springer-Verlag.

Bézivin, J. (2004). In search of a basic principle for model driven engineering. *Novatica Journal Special Issue*, *V*(2), 21–24.

Cetina, C., Fons, J., & Pelechano, V. (2008). Applying Software Product Lines to Build Autonomic Pervasive Systems. In *Proceedings of the 2008 12th International Software Product Line Conference*, (pp. 117–126). Washington, DC, USA: IEEE Computer Society.

Cetina, C., Giner, P., Fons, J., & Pelechano, V. (2009a). Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, *42*(10), 37–43.

Cetina, C., Giner, P., Fons, J., & Pelechano, V. (2010). Designing and prototyping dynamic software product lines: techniques and guidelines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, (pp. 331–345). Berlin, Heidelberg: Springer-Verlag.

Cetina, C., Haugen, O., Zhang, X., Fleurey, F., & Pelechano, V. (2009b). Strategies for variability transformation at run-time. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, (pp. 61–70). Pittsburgh, PA, USA: Carnegie Mellon University.

Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., MÃijller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., & Whittle, J. (2009). Software engineering for self-adaptive systems: A research roadmap. In B. Cheng, R. de Lemos, H. Giese, P. Inverardi, & J. Magee (Eds.) *Software Engineering for Self-Adaptive Systems*, vol. 5525 of *Lecture Notes in Computer Science*, (pp. 1–26). Springer Berlin / Heidelberg.

Clements, P., & Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc.

Cuadrado, J. S., Molina, J. G., & Tortosa, M. M. (2006). Rubytl: A practical, extensible transformation language. In A. Rensink, & J. Warmer (Eds.) *ECMDA-FA*, vol. 4066 of *Lecture Notes in Computer Science*, (pp. 158–172). Springer.

Czarnecki, K., & Kim, P. (2005). Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories At OOPSLA 2005*.

Dey, A. K. (2009). Modeling and intelligibility in ambient environments. *J. Ambient Intell. Smart Environ.*, *1*, 57–62.

Gomaa, H., & Shin, M. (2008). Multiple-view modelling and metamodelling of software product lines. *Software, IET*, *2*(2), 94 –122.

Hallsteinsen, S., Hinchey, M., Park, S., & Schmid, K. (2008). Dynamic Software Product Lines. *Computer*, *41*(4), 93 –95.

Hallsteinsen, S., Stav, E., Solberg, A., & Floch, J. (2006). Using Product Line Techniques to Build Adaptive Systems. In *Proceedings of the 10th International on Software Product Line Conference*, (pp. 141–150). Washington, DC, USA: IEEE Computer Society.

Haugen, Ø., Husa, K., Runde, R., & Sttølen, K. (2005). STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling*, *4*, 355–357. 10.1007/s10270-005-0087-0.

Haugen, Ø., & Sttølen, K. (2003). STAIRS - Steps to Analyze Interactions with Refinement Semantics. In *of Lecture Notes in Computer Science*, (pp. 388–402). Springer.

Horn, P. (2001). Autonomic computing: IBM's Perspective on the State of Information Technology.

Istoan, P., Nain, G., Perrouin, G., & Jezequel, J.-M. (2009). Dynamic Software Product Lines for Service-Based Systems. *Computer and Information Technology, International Conference on*, *2*, 193–198.

Jouault, F., & Kurtev, I. (2006). Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference, LNCS 3844*, (pp. 128–138). Springer.

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute.

Kephart, J., & Chess, D. (2003). The vision of autonomic computing. *Computer*, *36*(1), 41 – 50.

Kramer, J. (2007). Is abstraction the key to computing? *Commun. ACM*, *50*, 36–42.

March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decis. Support Syst.*, *15*(4), 251–266.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., & Cheng, B. H. (2004). Composing adaptive software. *Computer*, *37*, 56–64.

Miller, J., & Mukerji, J. (2003). MDA Guide Version 1.0.1. Tech. rep., Object Management Group (OMG).

Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., & Solberg, A. (2009). Models@ Run.time to Support Dynamic Adaptation. *Computer*, *42*, 44–51.

Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.-M., Solberg, A., Dehlen, V., & Blair, G. (2008). An aspect-oriented and model-driven approach for managing dynamic variability. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, & M. Völter (Eds.) *Model Driven Engineering Languages and Systems*, vol. 5301 of *Lecture Notes in Computer Science*, (pp. 782–796). Springer Berlin / Heidelberg.

Nance, R. E., Overstreet, C. M., & Page, E. H. (1999). Redundancy in model specifications for discrete event simulation. *ACM Trans. Model. Comput. Simul.*, *9*, 254–281.

OMG (2005). *MOF QVT Final Adopted Specification*. Object Modeling Group. ptc/07-07-07.

OMG (2006). *Meta Object Facility (MOF) Core Specification Version 2.0*.

Parra, C., Blanc, X., & Duchien, L. (2009). Context awareness for dynamic service-oriented product lines. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, (pp. 131–140). Pittsburgh, PA, USA: Carnegie Mellon University.

Plugin (2009). Feature modeling plug-in. http://gp.uwaterloo.ca/fmp/;. accessed november 2009.

Pure (2009). pure::variants. http://www.pure-systems.com/;. accessed november 2009.

Runde, R. K., Haugen, Ø., & Stølen, K. (2005). The pragmatics of stairs. In *FMCO*, (pp. 88–114).

Salehie, M., & Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, *4*, 14:1–14:42.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, *39*(2), 25–31.

Stahl, T., Voelter, M., & Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Sterritt, R., Parashar, M., Tianfield, H., & Unland, R. (2005). A concise introduction to autonomic computing. *Advanced Engineering Informatics*, *19*(3), 181–187.

Trinidad, P., Benavides, D., Ruiz-Cortes, A., Segura, S., & Jimenez, A. (2008). Fama framework. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, (p. 359).

Trinidad, P., Cortés, A. R., Peña, J., & Benavides, D. (2007). Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *International Workshop on Dynamic Software Product Lines*.

Vaishnavi, V., & Kuechler, W. (2004). Design research in information systems. http://desrist.org/design-research-in-information-systems.

Zhang, J., & Cheng, B. H. C. (2006). Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, (pp. 371–380). New York, NY, USA: ACM.

www.pros.upv.es

Centro de Investigación en Métodos
de Producción de Software
Universitat Politècnica de València
Camí de Vera s/n
Edificio 1F
46022 València
Spain
Tel: (+34) 963 877 007 (Ext. 83530)
Fax: (+34) 963 877 359

UNIVERSIDAD
POLITECNICA
DE VALENCIA