Applying Software Product Lines to Build Autonomic Pervasive Systems

Carlos Cetina Englada

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

A Thesis Presented in Partial Fulfillment of the Requirements for the Degree of M.Sc.

Centro de Investigación en Métodos de Producción de Software
Universidad Politécnica de Valencia
Camino de Vera s/n, E-46022, Spain

September of 2008

"Any customer can have a car painted any colour that he wants so long as it is black"

–Henry Ford,

My Life and Work, 1922

Contents

List of Figures

Chapter 1

Introduction

Software Product Line (SPL) engineering has proved itself as an efficient way to deal with varying user needs and resource constraints. However, the focus has been on the efficient derivation of customized product variants that once created, keep their properties throughout their lifetime.

The work developed in this Master Thesis proposes a SPL method to develop *dynamically-adaptive pervasive systems* in a systematic manner. This approach allows pervasive systems to use the variability modeling from the SPL design at run-time. The approach makes use of two primary ideas: (i) collective modeling instead of individual modeling; and (ii) the application of product-line architectures (PLAs) [1]. This work extends the production operation of the SPLs in order to augment SPL products with variability models and also some extra assets that enable reconfiguration. It makes use of the variability models and the available resources to find the "best" reconfiguration of the software system to achieve the user goals. These variability models assist the execution strategy to determine the steps that are necessary to reconfigure the software system. Then, the PLA is rapidly retargeted to the desired configuration. The use of variability models at run-time enables the pervasive system to dynamically decide how to achieve the user goals in an efficient manner.

This adaptation approach is focused on two adaptation scenarios very common in Pervasive Systems: evolution (a resource is added) and involution (a resource is removed). These scenarios have different requirements regarding adaptation, and

the way in which models are handled at run-time should consider those particular requirements. A model-based approach is introduced in order to organize the knowledge required for adaptation according to the specific demands of each adaptation scenario. In involution scenarios, we use models with precalculated knowledge in order to provide an autonomic response in a reduced amount of time. While in evolution scenarios, we offer an advanced system response (feature dependency resolution and user participation) because we consider that installing new resources in the system is not as critical as handling resource failures.

This work has used dynamic adaptation in a manner of constrained adaptation for the development of autonomic pervasive systems. By constrained adaptation we refer to the fact, that we require complete specifications of the adaptation behavior already at design time, which are evaluated by the system at runtime in order to adapt to the current runtime situation. Since the models forming the basis for the adaptation behavior are available at design time, we are able to conduct thorough analysis of the specifications for the purpose of validation and verification. We are able to guarantee deterministic adaptation behavior at runtime, which is essential for reliable systems.

1.1 Motivation

Increasingly, software needs to dynamically adapt its behavior at run-time in response to changing conditions in the supporting computing infrastructure and in the surrounding physical environment [2]. Adaptability is emerging as a necessary underlying capability, particularly for highly dynamic systems. Pervasive systems are highly dynamic and fault-prone since new kinds of entities (sensors, actuators, external software systems) can enter these systems at any time. Existing entities may fail or leave the system for a variety of reasons: hardware faults, OS errors, software bugs, network faults, etc. The dynamic and fault-prone nature of these systems makes it necessary to design new techniques to ensure their smooth operation.

Pervasive computing is defined as a technology that “weaves itself into the fabric of everyday life until it is indistinguishable from it” [3]. To be successful, the per-

vasive computing functioning should be transparent to the user. Such transparency is achievable if the software frees users from having to repair and reconfigure the system when faults or changes occur in the environment.

Autonomic systems [4] configure themselves automatically in accordance with high-level policies (self-configuration) and can detect, diagnose, and repair localized problems (self-healing). In a car or an air-plane, faults need to be be repaired without shutting down and restarting the entire system. In a smart home, end-users should be able to perform homes upgrades (install new sensors or actuators) without having to reconfigure the software system. To achieve this autonomic goal, a pervasive system needs to "know itself", and its components should also possess a system identity [5]. Pervasive systems need to evolve in an autonomic way even though they are built using a Software Product Line (SPL) approach.

Many research efforts have proposed ways for *automating variant construction* from component-based models or feature models. Some of these efforts are focussed on providing techniques for reasoning about the best variant selection according to a set of requirements [6, 7]. Other techniques also address the problem of producing and deploying the calculated variant from the SPL [8, 9].

There is not a lack of adaptive pervasive system but current approaches address the design of the dynamic reconfiguration in an ad hoc manner. There is a need for a **Systematic Software Engineering Approach** which models possible configurations of an application as a product family capable of automatically reconfiguring from one configuration of the family to another. Furthermore, **Run-time Adaptation** must be capable to compensate failures as far as possible.

1.1.1 A Systematic Software Engineering Approach

Since the specification of the adaptation behavior is a complex and error prone task, a systematic software engineering approach for the development of such systems is required. Such constructive methodological support involves a dedicated methodology enabling developers to systematically develop adaptive embedded systems.

First, this includes a seamless modeling methodology. In this regard, it is important to make the complexity manageable, e.g. by separating the adaptable spec-

ification of the non-adaptable specification.

Second, the seamless software engineering approach also includes the model-based analysis, validation and verification of dynamic adaptation. For dependable pervasive systems, it is indispensable to have a means to analyze the adaptation behavior already at design time and to guarantee certain properties.

1.1.2 Run-Time Adaptation

There exist different means, like fault prevention, fault tolerance, fault removal and fault forecasting, to reach higher reliability in current software engineering approaches. These means certainly help to amend those characteristics in software systems, but are not sufficient in pervasive systems where high dynamics come into play and/or failures must be compensated at runtime.

Two possible approaches exist to address this problem: a) provide sufficient redundancy by means of additional (physical) devices or b) make systems adaptive so that they are capable to compensate failures by runtime adaptation. Whereas the first approach provides the higher degree of dependability, the second approach is less expensive and brings also the advantage of making systems more flexible with respect to other QoS attributes (i.e. to react to varying resources or changing user goals).

Moreover dynamic adaptation is the more flexible approach so that it is also easily possible to realize conventional safety and reliability patterns based on dynamic adaptation. In the case of errors, the system adapts to compensate these errors as far as possible. In some cases it is accepted that an error leads to a degraded functionality as long as the safety of the system is ensured (graceful degradation or survivability [18]).

1.2 Problem Statement

The development of Autonomic Pervasive Systems is not a closed research topic. We can see from the above discussion how some problems need to be still considered. The work that has been done in this thesis tries to improve the development of Autonomic

Pervasive Systems by considering these problems. In particular, the problems that this master thesis tries to solve can be stated by means of the following two problem statements:

For the development of pervasive systems based on dynamic adaptation, it is indispensable to come to a seamless software engineering approach: an integrated methodology guiding the developer systematically from the requirements to an adaptive system. Since the specification of the adaptation behavior is a complex and error prone task, a systematic software engineering approach for the development of such systems is required. It is necessary that the adaptation behavior is explicitly defined. However, such constructive methodological support for the development of safe and reliable embedded systems is still in its infancy.

1.3 Contributions

In this master thesis, we present a SPL approach for developing autonomic pervasive systems in a systematic manner. This approach allows pervasive systems to use the variability modeling from the SPL design at run-time. This approach is developed by achieving the following goals: (a) Identifying the suitable variability knowledge for performing adaptation; (b) Extending the SPL to transfer the variability knowledge to the SPL products; and (c) Augmenting the SPL products to reuse the variability knowledge at run-time in order to perform the adaptation.

The main contributions of this thesis are developed in order to address the problem statement presented above. In particular:

1. In our approach not only an execution platform or mechanism to realize dynamic adaptation at runtime is provided, but also a dedicated methodology enabling developers to systematically develop autonomic pervasive systems.

2. Therewith this model-driven approach makes it possible to identify reasonable configurations in an early stage of the development process without first implementing them. Furthermore, it also benefits from the whole range of typical gains brought by model-driven engineering approaches (i.e. validation, verification, reuse, automation).

3. Our approach separates the model specifying the non-adaptive behavior from the one specifying the adaptive behavior, thus making the model more amenable to human inspection and automated analysis.

4. The differences between evolution (a resource is added) and involution (a resource is removed) scenarios are addresses in our approach. In involution scenarios, we use models with precalculated knowledge in order to provide an autonomic response in a reduced amount of time. While in evolution scenarios, we offer an advanced system response (feature dependency resolution and user participation) because we consider that installing new resources in the system is not as critical as handling resource failures.

5. Since the models forming the basis for the adaptation behavior are available at design time, we are able to guarantee deterministic adaptation behavior at runtime, which is essential for reliable systems.

Furthermore, we have elaborated a taxonomy of SPLs for adaptive products. We intend to summarize the SPL architectures that have been proposed at date, dividing them in connected and disconnected SPL depending on their dependence with the SPL infrastructure.

Finally, the Moskitt Feature Modeler tool (MFM) has been developed in the context of this master thesis. MFM is a feature model editor where the tri-state configuration proposed in this work has been implemented. In a tri-state configuration, features can be set to one of the following states: discarded, deactive or active. Active features conform the initial configuration of the pervasive system, whereas deactive features identified the quiescent components of the system.

1.4 Outline

The rest of this thesis is organized as follows:

Chapter 2: Background. This chapter introduces the background (Software Product Lines, Model Driven Development and Autonomic Computing) on top of which the remaining chapters are developed.

Chapter 3: Related Work. This chapter presents a critical analysis of the most well known SPL approaches for the development of adaptive pervasive systems. In particular, we focus on where the adaptation is performed in these approaches (whether in the SPL or in the system itself). We also study these approaches form the following perspectives: autonomic degree, adaptation capabilities and computational overload.

Chapter 4: A SPL-Based Approach for Building Dynamically Reconfigurable Systems. This chapter introduces the SPL method for developing dynamically-adaptive pervasive systems. First, we describe suitable adaptation scenarios where the SPL knowledge can be applied. Then, we present the SPL extensions to transfer this knowledge to the SPL products.

Chapter 5: The Reconfiguration Realization in both SPL and Products. In this chapter, we analyze the reconfiguration process both from the point of view of the SPL and the Products. Finally, we describe the key infrastructure components to support the reconfiguration process.

Chapter 6: Reconfiguration Architecture. In this chapter, first we discuses design patterns for adaptation architectures and then the architecture to realize dynamic adaptation at runtime is presented. This architecture supports different adaptation mechanisms depending on how much critical the adaptation is.

Chapter 7: Case Study: An Autonomic Smart Home. We illustrate the proposed SPL for Autonomic Pervasive Systems by modeling a smart home family: a localized technology-augmented environment where people perform everyday life activities. This chapter presents the models of the SPL and a set of adaptation-scenarios to check the adaptation capabilities.

Chapter 8: Conclusions. This chapter concludes by discussing the appropriateness of the solution proposed in this thesis. Contributions are summarized and future works are proposed.

Chapter 2

Background

2.1 Abstract

Software Product Lines (SPL) Software Product Line (SPL) engineering intends to produce a set of products that share a common set of assets in an specific domain. SPL engineering techniques allow to adapt a product to the customer needs while its production costs and time to market are decreased. SPL promotes the shift from the development of an stand-alone programs to the development of a family of programs.

Model Driven Development (MDD) is a paradigm where we can construct a model of a software system that we can then transform into the real thing. The goal of this paradigm is to automatically translate an abstract specification of the system into a fully functional software product.

Autonomic Computing is an initiative started by IBM in 2001. Its ultimate aim is to develop computer systems capable of self-management, to overcome the rapidly growing complexity of computing systems management, and to reduce the barrier that that complexity poses to further growth.

Pervasive Computing is a post-desktop model of human-computer interaction in which information processing has been thoroughly integrated into everyday objects and activities. As opposed to the desktop paradigm, in which a single user consciously engages a single device for a specialized purpose, someone "using" ubiquitous computing engages many computational devices and systems simultaneously.

This chapter provides a quick glance at Software Product Lines, Model Driven Development, Autonomic Computing and Pervasive Computing.

2.2 Software Product Lines

Mass production was popularized by Henry Ford in the early 20th Century. McIlroy coined the term software mass production in 1968 [10]. It was the beginning of Software Product Lines. In 1976, Parnas introduced the notion of software program families as a result of mass production [11]. The use of features (to drive mass production) was proposed by Kang in the early 1990s [12]. Shortly, the first conferences appeared turning SPL into a new body of research [13, 14].

2.2.1 Definition

SPLs are defined as "a set of software-intensive systems, sharing a common, managed set of features that satisfy 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" [15]. This definition can be redefined into five major issues:

1. **Products**. a set of software-intensive systems.... SPL shift the focus from single software system development to SPL development. The development processes are not intended to build one application, but a number of them (e.g., 10, 100, 10,000, or more). This forces a change in the engineering processes where a distinction between domain engineering and application engineering is introduced. Doing so, the construction of the reusable assets (platform) and their variability is separated from production of the product-line applications.

2. **Features**. ...sharing a common, managed set of features.... Features are units (i.e., increments in application functionality) by which different products can be distinguished and defined within an SPL [16].

3. **Domain**. ...that satisfy the specific needs of a particular market segment or mission.... An SPL is created within the scope of a domain. A domain is a

specialized body of knowledge, an area of expertise, or a collection of related functionality [17].

4. **Core Assets**. ...are developed from a common set of core assets.... A core asset is "an artifact or resource that is used in the production of more than one product in a software product line" [15].

5. **Production Plan**. ...in a prescribed way. It states how each product is produced. The production plan is "a description of how core assets are to be used to develop a product in a product line and specifies how to use the production plan to build the end product [18]. The production plan ties together all the reusable assets to assemble (and build) end products. Synthesis is a part of the production plan.

2.2.2 Background

According to Clements and Northrop, a software product line consists of a set of software products sharing a common set of features that satisfy the needs of a particular domain and that are developed from a common set of core assets in a prescribed way. Therefore, software product line engineering is about producing families of similar systems rather than the production of individual systems. Software product line engineering consists of three main activities: domain engineering (also called core asset development) and application engineering(also called product development) and management. These three activities are complementary and provide feedback to each other. Domain engineering deals with the production of software assets to be used in different products of the product line. On the other hand, application engineering deals with the production of individual systems from core assets and individual needs and management that is responsible for giving resources, coordinating, and supervising domain and application engineering activities.

In general, facing an SPL implies to distinguish between two separate processes, namely, the domain engineering process, and the application engineering process. Domain Engineering is defined as the activity of collecting, organizing and storing past experience in building systems or parts of systems in a particular domain in

the form of reusable assets (e.g., architecture, models, code, and so on), as well as providing an adequate means for reusing these assets (...) when building new systems [19].

Using a "design-for-reuse" approach, domain engineering (core asset development [15]) is in charge of determining the commonality and the variability among product family members. In general, domain engineering is divided into domain analysis, domain design and domain implementation. However, this simple division hides a number of practices and activities. Refer to [15, 20] for a complete account.

Application Engineering is the process of building a particular system in the domain [19]. Application engineering (a.k.a., product Development [15]) is responsible for deriving a concrete product from the SPL using a "design-withreuse" approach. To attain this, it reuses the reusable assets developed previously. This process is subdivided into application analysis, application design and application implementation (some other activities are omitted as well [15]). Some authors introduce a separated process for Management where organizational issues are handled specifically [15].

2.3 Model Driven Development

2.3.1 Definition

The arrival of the MDSD and MDA are changing the way of using models in the development of software. As stated by Agrawal [3]:

> the models are not merely artifacts of documentation, but living documents that are transformed into implementations. This view radically extends the current prevailing practice of using UML: UML is used for capturing some of the relevant aspects of the software, and some of the code (or its skeleton) is automatically generated, but the main bulk of the implementation is developed by hand. MDA, on the other hand, advocates the full application of models, in the entire life-cycle of the software product.

The goal of these methods is to automatically translate an abstract specification of the system into a fully functional software product.

2.3.2 Background

Model-Driven Software Development (MDSD) is the notion that we can construct a model of a software system that we can then transform into the real thing [21]. Models have been used for a long time in the software development field. From formal and executable specification languages (like OBLOG [22], TROLL [23] or OASIS [24]), which have not been widely accepted by the industry, to the most accepted notations (like UML [25]) and processes (like RUP [26]) models are present in the software development area.

Kent defines Model Driven Engineering (MDE) by extending MDA with the notion of software development process (i.e., MDE emerged later as a generalization of the MDA for software development) [27]. MDE refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle4. Kurtev provides a discussion on existing MDE processes [28] (refer to [29, 30] for a specific approach). In general, these approaches introduce concepts, methods and tools [31]. All of them are based on the concept of model, meta-model, and model transformation.

Model Driven Architecture (MDA) is a concrete realization of MDD. As mentioned above,MDA classifies models into 2 classes: Platform Independent Models (PIMs) and Platform Specific Models (PSMs) [32]. A PIM is a view of a system from a platform-independent viewpoint. Likewise, a PSM is a view of a system from a platform-dependent viewpoint [32]. Doing so, the definition of platform becomes fundamental.

2.4 Autonomic Computing

In October 2001, IBM released a manifesto [4] 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.

2.4.1 Definition

> Autonomic Computing is a potential solution to the problem of increasing system complexity and costs of maintenance. It is an approach where the ultimate goal is to create computer systems that can manage themselves while hiding their complexity from the end users.

Thus, by hiding and removing the low-level complexities from the realm of human control, humans can be freed from the burdens of maintenance to concentrate on achieving higher-level, business orientated objectives.

The concept and vision behind Autonomic Computing is certainly a promising one. However, due to its goal orientated approach, development in the Autonomic Computing field is driven by goal achievement and not by systematic engineering processes. As a result, a commonly accepted definition of Autonomic Computing is yet to be established, allowing any technology that exhibits behaviours of self-management to automatically be classified as Autonomic Computing.

2.4.2 Background

In Kepharts and Chess Vision of Autonomic Computing [33] the following four system abilities are discussed:

- **Self-configuring** involves automatic incorporation of new components and automatic component adjustment to new conditions.
- **Self-optimising** on a system level is about automatic parameter tuning on services. A suggested method to do this is explore, learn and exploit.

- **Self-healing** from bugs and failures can be accomplished using components for detection, diagnosis and repair.

- **Self-protecting** of systems will prevent large-scale correlated attacks or cascading failures from permanently damaging valuable information and critical system functions. It may also act proactively to mitigate reported problems.

It is claimed that as these aspects become properties of a general architecture they will merge into a single self-maintenance quality. The architecture of an Autonomic Computing system will be a collection of components called autonomic elements , which encapsulates managed elements. The managed element can be hardware, application software or an entire system. An autonomic element is an agent, managing its internal behaviour and relationships with others in accordance to policies. It is driven by goals, by other elements or by contracts established by negotiation with other elements. System self-management will arise both from interactions among agents and from agents internal self-management.

An autonomic multi-agent system will run in agent-oriented architectures of yet unknown proportions. These architectures present numerous engineering challenges to be solved, involving agent lifecycles and relationship management such as negotiation and trust, just to mention a few. There are also scientific challenges such as induction of global behaviour, control theories, machine learning, etc.

At IBM, the Autonomic Computing initiative spans across all levels of computer management, from the lowest levels of hardware to the highest levels of software systems. IBM System Journal has collected some of this work. On the hardware level, systems are dynamically upgradable [34]. On the operating system level, active operating system code is replaced dynamically [35]. Some work on autonomic middleware can be found at other sources [36] [37] [38]. On the application level, databases self-validate optimisation models [39] and web servers are dynamically reconfigured by agents to adapt service performance [40].

These examples illustrate that, although some of the characteristic Autonomic Computing features seem to be far away, some of the ideas from Autonomic Computing have already been put into practice. In the next section, we will examine other current technological signs of a coming new computing era.

2.5 Pervasive Computing

2.5.1 Definition

> The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.

This can be the summary of what Weiser stated as Ubiquitous Computing in his seminar paper [41]. This vision is based on the construction of computing saturated environments properly integrated with human users. Essential to the vision is networking [42], for without the ability of these computing devices to communicate with one another, the functionality of such a system would be extremely limited. Weiser stated that the next generation computing environment would be one "in which each person is continually interacting with hundreds of nearby wirelessly connected computers". At the time, such forms of wireless networking were still in their infancy, but today with wireless Wi-Fi, WiMax and Bluetooth, the possibilities for such dense local area networks are entering the realm of commercial reality.

2.5.2 Background

Several terms are used in literature for talking about similar concepts. The main differences depends on the context of use: for instance, EEUU vs Europe or Academy vs Industry. According to Mattern [43], while Weiser saw the term "ubiquitous computing" in a more academic and idealistic sense as an unobtrusive, human-centric technology vision that will not be realized for many years yet, industry (IBM) has coined the term "pervasive computing" with a slightly different slant [44, 45].

In [42] is stated that while researchers in the United States were working on the vision of ubiquitous computing, the European Union began promoting a similar vision for its research and development agenda. The term adopted in Europe is "ambient intelligence" (coined by Emile Aarts of Philips) which has a lot in common with Weiser's ubiquitous computing vision. This point of view is confirmed by the great number of events and research projects that are organized and/or funded in

Europe under this term which topics clearly matches the ones that are inside the scope of the ubiquitous computing area.

Although subtle differentiations could be done between these terms according to their etymological meanings (nor ubiquitous implies intelligence, neither intelligence implies pervasiveness, etc.), in general we can state that the main idea or vision behind them is the same and, therefore, they can be equally used.

Chapter 3

Related Work

3.1 Abstract

This chapter presents a critical analysis of the most well known SPL approaches for the development of adaptive pervasive systems. In particular, we focus on where the adaptation is performed in these approaches (whether in the SPL or in the system itself). In this chapter we intend to summarize the SPL architectures that have been proposed at date, dividing them in connected and disconnected SPL depending on their connectivity and dependence with the SPL development infrastructure. We also study these approaches form the following perspectives: autonomic degree, adaptation capabilities and computational overload. Finally, We discuss the benefits of both connected and disconnected SPL architectures, giving some details about its internals.

Before presenting this analysis some background about Adaptation Frameworks without a Systematic Engineering Support is introduced.

3.2 Adaptation Frameworks

Most of the research of recent years has been focused on explicit adaptation without an underlying engineering of adaptation. The main characteristic shared by these approaches is the presence of a dedicated runtime adaptation framework. This framework could be a central component in the system coordinating all adapta-

tion processes or it could be a decentralized aspect that is scattered to different components. In any case, however, the dynamic adaptation is explicitly controlled and/or coordinated. The adaptation frameworks are usually quite simple and require a model or specification telling them under which condition which adaptation strategy has to be chosen. For complex systems it is hardly possible to define such a specification ad hoc without applying an appropriate, constructive development methodology. Therefore this leads to another challenge, an immense effort is required to manage the complexity of the adaptation behavior. A vast majority of the researches concerning constraint adaptation are dealing with the development of runtime adaptation frameworks ([46], [47], [48], [49] to name a few examples).

A project of Carnegie Mellon University called RoSES [47] uses product family architectures for realizing dynamic reconfiguration. They define a product family and, in the case of faults they reconfigure the system to an alternative product configuration from this product family. [46] uses a low-fidelity high-speed search algorithm and a high-fidelity search algorithm to determine the next system configuration. If a reconfiguration is subject of hard timing constraints, e.g., if failures occur that affect critical system services, the high speed algorithm searches for a viable configuration that implements all critical functions. The high-fidelity algorithm that searches high utility system configurations is applied when no timing constraints are given, e.g., a viable configuration is currently active.

In [48] so-called Containment Units monitor the quality of functionalities. Depending on the detected quality, the containment unit turns the functionalities off or replaces them with alternative functionalities.

The researches of the embedded adaptive systems laboratory EASL [50] deals with the transitions between configurations, taking into account that the configurations might have continuous or discrete states. Although the EASL does not aim at the development of a framework their research results contribute to evolution stage 2, because they take the specification of the adaptation behavior for granted and focus on the realization of the adaptation behavior.

In [51], the authors introduce a method for constructing and verifying adaptation models using Petri nets. In [52], linear temporal logic is extended with an

adapt operator for specifying requirements on the system before, during and after adaptation. An approach to ensure correctness of component based adaptation was presented in [53], where theorem proving techniques are used to show that a program is always in a correct state in terms of invariants. [54] introduces a formal model of reconfiguration and an associated set of high-level, general system dependability properties that can be verified.

Although these approaches already support verification of dynamic adaptation, the current state of the research is at the very beginning of a software engineering of adaptive systems. The main reason for this is that they are based on adaptation behavior specifications without providing adequate constructive modeling methodologies. Therefore these specifications are hardly to define in a reasonable manner for real systems. For instance specifying adaptation behavior using Petri nets [51] is not an intuitive way to design complex industry sized systems like the ESP (Electronic Stability Program). Furthermore, in current researches the quality assessment of adaptive systems is solely based on verification techniques, however, other techniques like probabilistic analysis are indispensable for quality assurance in particular with respect to assurance of safety requirements.

As an example the MARS project aims at providing a seamless engineering approach from the requirements to running systems. MARS uses dynamic adaptation as a flexible error handling technique aiming at cost-efficient development of dependable embedded systems. Starting from a Feature-Model [55] the system architecture is defined using the Mars modeling language [56], which is basically an extension of established concepts of architecture description languages [57]. For this purpose we use the modeling environment GME [58].

From the analysis model that specifies the adaptation behavior, a design model in Matlab/Simulink is generated that combines adaptation and functionality in an integrated model building the basis for the subsequent system design [59]. The validation and verification techniques of the adaptation behavior include simulation, verification [60], and probabilistic analysis [61].

3.3 Dynamic Software Product Lines

SPL main objective is producing products while costs and time-to-market are reduced by an intensive reuse of commonalities and a suitable variability management. Products are commonly produced by selecting the features that are part of a product and removing those that are not part of it. To make this decision, features are selected and/or discarded at different binding times. Those features thought to be bound at runtime are kept in the final product even when they may not be used by the final product. The product must provide the mechanisms to select the suitable feature at runtime and optionally reconfigure the product. After the production, no automated activity is specified in SPL development to maintain a product in connection with the SPL so it may not eventually benefit from feature updates.

On the other hand, DSPL development mainly intends to produce configurable products [62] whose autonomy allows to reconfigure themselves and benefit from a constant updating. 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 analyser 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 runtime 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 runtime. As a consequence, new features may be added to an existing product or even existing features may be updated at runtime.

Comparing both, SPL and DSPL development, DSPL development might be considered as a particular case of SPL where following properties are added:

- Adaptation to changing requirements and environment: a product is prepared to response to the so called *adaptation triggers* which alert the system to a change in the environment or the conditions the product is working in.

Besides, a user may explicitly ask for a change in a product configuration. Both cases are analysed for their consequences and if it is possible, the system is reconfigured to adapt itself to new situations.

- Autonomic capabilities: a CP is able to make decisions about the features that are activated and deactivated at a time from the information obtained from its environment and whenever an adaptation trigger or an user request arrives to the decision maker.

- Product Updates: as all the features are bound at runtime, updating an existing product with new features or updates is eased

Several approaches for developing CP using DSPLs have been presented along the published literature. In this work, we have classified these approaches in two categories, according to the way in which product adaptation is considered. These categories are the following:

- **Connected DSPL.** The DSPL is in charge of the product adaptation.

- **Disconnected DSPL.** The CP itself is in charge of the product adaptation.

We describe both connected and disconnected DSPLs from the *product perspective*. In this perspective, we study the advantages and disadvantages obtained as result of incorporating adaptation in SPL products. The following criteria help us to evaluate the return of investment on DSPLs:

- **Autonomic degree.** This criterion addresses how much products depend on the DSPL to perform adaptation.

- **Adaptation capabilities.** This criterion addresses the adaptation level achieved by the CP.

- **Computational overload.** This criterion addresses how much computational overload is introduced by the DSPL approach in the CP execution.

Figure 3.1 shows the methods included in connected and disconnected categories. They have been ordered according to year in which they appear in the literature.

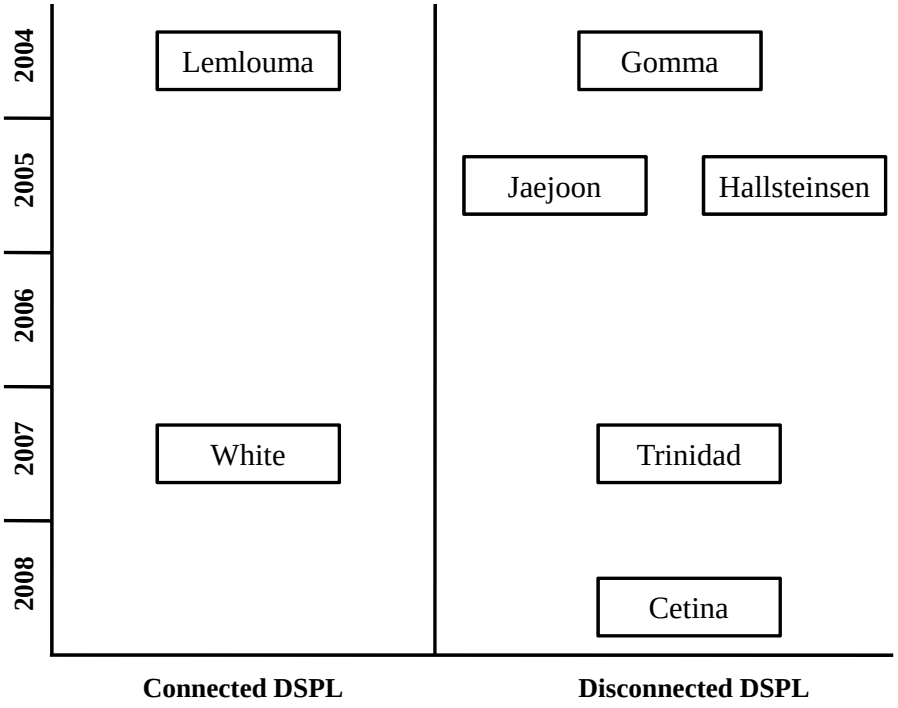

Figure 3.1: Classification of DSPL

According to Figure 3.1, there are six DSPL that focus their efforts on developing configurable products. An overview of these DSPL is presented next:

3.3.1 Connected SPL

Connected DSPLs stay in touch with products in order to send them updates. These updates enable products to deal with context changes. Figure 3.2 shows the steps to send the updates from the DSPL to the CPs.

1. The CP senses a relevant change which starts the adaptation process. Both changes in the environment and in the CP itself can trigger the adaptation process.

2. The CP sends information about the change to the SPL. Optionally, the CP can locally preprocess the information in order to send a more specific information to the SPL.

3. The SPL incorporates the acquired information to the product requisites and then it calculates a new CP variant.

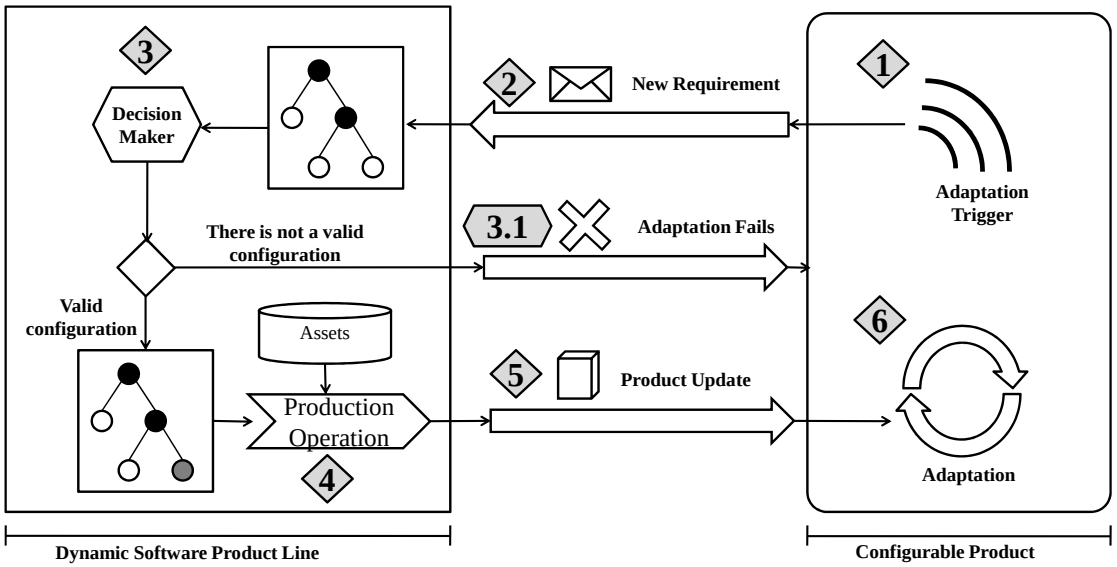

Figure 3.2: Connected DSPL Overview

(a) If there is no variant that satisfies the product requisites, then the SPL notifies the CF and the adaptation process fails.

4. The SPL generates the CP update. This update can be the whole calculated variant or the difference between the old variant and the new one.

5. The SPL sends the update to the CP.

6. The CP updates itself using the update information from the SPL.

According to the criteria presented before, we characterize a connected DSPL as follows:

- **Autonomic degree.** The system depends on the SPL availability in order to get the system **updates** to perform the adaptation.

- **Adaptation capabilities.** To address adaptation, variability knowledge indicates the involved components. However, some of these components are not in the system. In this case, the system has to get these components from the SPL. Hence, it is necessary a bidirectional connection between the DSPL and the CP. If this connection becomes unavailable then the adaptation can not be performed.

- **Computational overload.** An disconnected DSPL approach introduces the following additional overload in the CP execution: (1) the communication with the SPL (to get system updates) and (2) the on-line installation of updates.

Lemlouma. Lemlouma et al. [8] present Negotiation and Adaption Core architecture for adapting and customizing content before delivering it to a mobile device. Their strategy takes into account device preferences and capabilities which are specified using a device independent model. These models are queried using the XQuery language and the adaptation is achieved by means of client repositories and SOAP services.

Adaptation Trigger. There is a real time evaluation of the context dimensions. Lemlouma explicitly identifies the following context dimensions: user preferences, network speed and current confection protocol.

White. The Scatter tool [9] was developed by White et atl. to address efficient online variant selection. Scatter captures the requirements of the product line architecture and the resources of a mobile device and then quickly constructs a custom variant for the device. This tool also ensures that variant selection is optimal with regard to a configurable cost function.

Adaptation Trigger. A mobile device discovery service obtains the non-functional properties of a devices such as *JVMVersion* or *Position*. This service is implemented using SOAP-based web service and a CORBA remoting mechanism for remotely communicating device characterizations as they are discovered.

3.3.2 Disconnected SPL

Disconnected DSPLs produce CP which can reconfigure itself to deal with contextual changes. Compared with connected DSPLs, CP reconfigures itself without any DSPL contact. CP are augmented with variability knowledge and quiescent components in order to perform the reconfiguration as Figure 3.3 shows:

1. The CP senses a relevant change which starts the adaptation process. Both changes in the environment and in the CP itself can trigger the adaptation process.

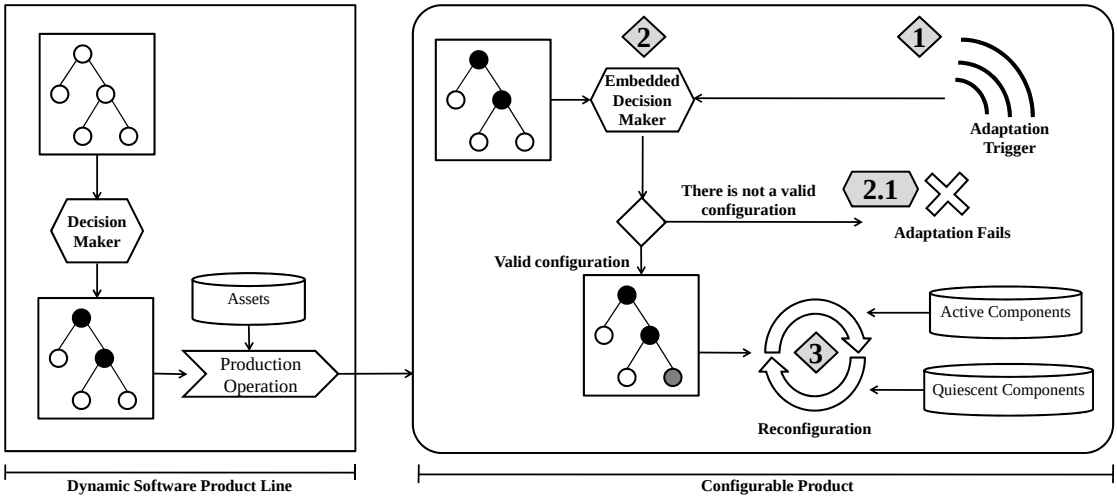

Figure 3.3: Disconnected DSPL Overview

2. The CP calculates a new configuration to deal with the sensed change.

   (a) If there is no configuration that satisfies the product requisites, then the adaptation process fails.

3. The CP reconfigures itself to apply the calculated configuration. The reconfiguration operation implies (1) start/stop components and (2) establish connections between them.

According to the criteria presented before, we characterize a disconnected DSPL as follows:

- **Autonomic degree.** The CP have no dependency of the SPL to perform the adaptation because there is necessary no connection between the SPL and the CP. The adaption depends only on CP resources. The CP **reconfigures** itself to perform adaptation.

- **Adaptation capabilities.** In general, the more variability knowledge the system has about itself and its variants, the more adaptable the system will get. This knowledge is captured in the variability models incorporated to the system. However, the variability models must be complemented with system components. Some components conform the initial system configuration, while the others are used in system reconfiguration. In conclusion, the adaptation

capabilities depends on the knowledge captured in the models and on the number of components for system reconfiguration.

- **Computational overload.** A connected DSPL approach introduces a computational overload to the system execution when the adaptation is triggered. This overload comes from (1) the model queries and (2) the execution of the reconfiguration (starting stopping and linking system components).

Gomaa. Reconfigurable Product Line UML Based Environment (RPLUSEE) [63] was proposed by Gomaa et al. Their main contribution is provisioning software dynamic reconfiguration patterns. Depending on the location of dynamic reconfiguration information, these patterns are classified into master-slave, centralized, client-server and decentralized. This method also provides reconfiguration Statechart and reconfiguration transaction models for the dynamic reconfiguration. This approach focuses on high-level specifications of dynamic reconfigurable units; however, it does not describe techniques and guideline for for reconfigurable component identification, design and implementation detail.

Adaptation Trigger. Users specify runtime configuration changes so that executable system is dynamically changed from the old configuration to the new configuration.

Lee. Lee et al. proposed a systematic method to developing dynamically reconfigurable core assets and a reconfigurator that monitors and manages product configuration at run-time [64]. The method first analyzes a product line in terms of features and their binding time. Then, core assets are developed with the analysis results as key design driver. Finally, the developed reconfigurator address reconfiguration contexts, reconfiguration strategies and reconfiguration actions (what to do to reconfigure)

Adaptation Trigger. The configuration plane is in charge of detecting contextual changes. The plane consists of two components: *Master Configurator* and *Local Configurator*. *Master Configurator* collects information from *Local Configurators* and/or external probes to detect contextual changes. Each *Local Configurator* monitors local messages within the product. **Hallsteinsen.** The MADAM approach [65]

was developed by Hallsteinsen et al. This approach builds adaptive systems as component based systems families with the variability modeled explicitly as part of the family architecture. MADAM uses property annotations on components to describe their Quality of Service. For example a Video Streaming component may have properties such as start up time, jitter and frame drop. At run-time, the adaptation is performed using these properties and a utility function for selecting the component that best fits the current context.

Adaptation Trigger. The *Context Manager* is responsible for managing and monitoring a set of contexts in the system environment relevant for the adaptation. Context includes execution platform context elements such as network and memory resources, the environment context elements such as light and noise, and user context elements location and stress level.

Trinidad. Trinidad et al [66] present a process to automatically build a component model from a feature model based on the assumption that a feature can be modeled as a component. This process focuses on enabling a dynamic SPL to dynamically changing a product by activating or deactivating its features at run-time.

Adaptation Trigger. One o more users set the requirements of the product.

3.4 Conclusions

We have presented a brief analysis of the main approaches published in the literature that provide support for the development of adaptive systems. In this analysis, we have focused on the adaptation techniques that these approaches propose. Then, we have briefly overview first those approaches based on adaptation frameworks but lacks of an underlying systematically method. Finally, we present a critical analysis of the most well known SPL approaches for the development of adaptive pervasive systems. In particular, we focus on where the adaptation is performed in these approaches, whether in the SPL (Connected SPL) or in the system itself (Disconnected SPL). Table 3.1 summarizes the specification techniques of each the analyzed approaches as well as the underlying infrastructure to support adaptation.

	Variability Specification	*Adaptation Infrastructure*
Gomaa	UML State Charts	Reconfiguration Patterns
Lemlouma	Device Independent Model	SOAP Services
Lee	Feature Model with Binding Units	Dynamically Reconfigurable Assets
Hallsteinsen	UML QoS Properties Profile	Planning Process
White	Requirements and Resources Specification	Variant Delivery
Trinidad	Feature Model	Relationship Components
Cetina	Scope, Commonalities and Variability Model	OSGI Services

Table 3.1: DSPL Comparison

Chapter 4

A SPL-Based Approach for Building Dynamically Reconfigurable Systems

4.1 Abstract

In this chapter, we propose a model-driven SPL for developing autonomic pervasive systems. The process focusses on reusing the SCV knowledge from the SPL design to the SPL products. This SCV knowledge enables SPL products to deal with evolution in an autonomic way. First, we describe suitable adaptation scenarios where the SPL knowledge can be applied. Then, we presente the SPL extensions to transfer this knowledge to the SPL products.

4.2 Reconfiguration Scenarios

Pervasive systems can be found in numerous and heterogeneous domains such as smart homes, health care, mobile devices, automotive, emergency situations and urban domains. All of these domains are composed of different entities and have different evolutionary needs.

In this section, we propose a pervasive systems categorization from the evolutionary point of view how pervasive systems respond to changes in their environment.

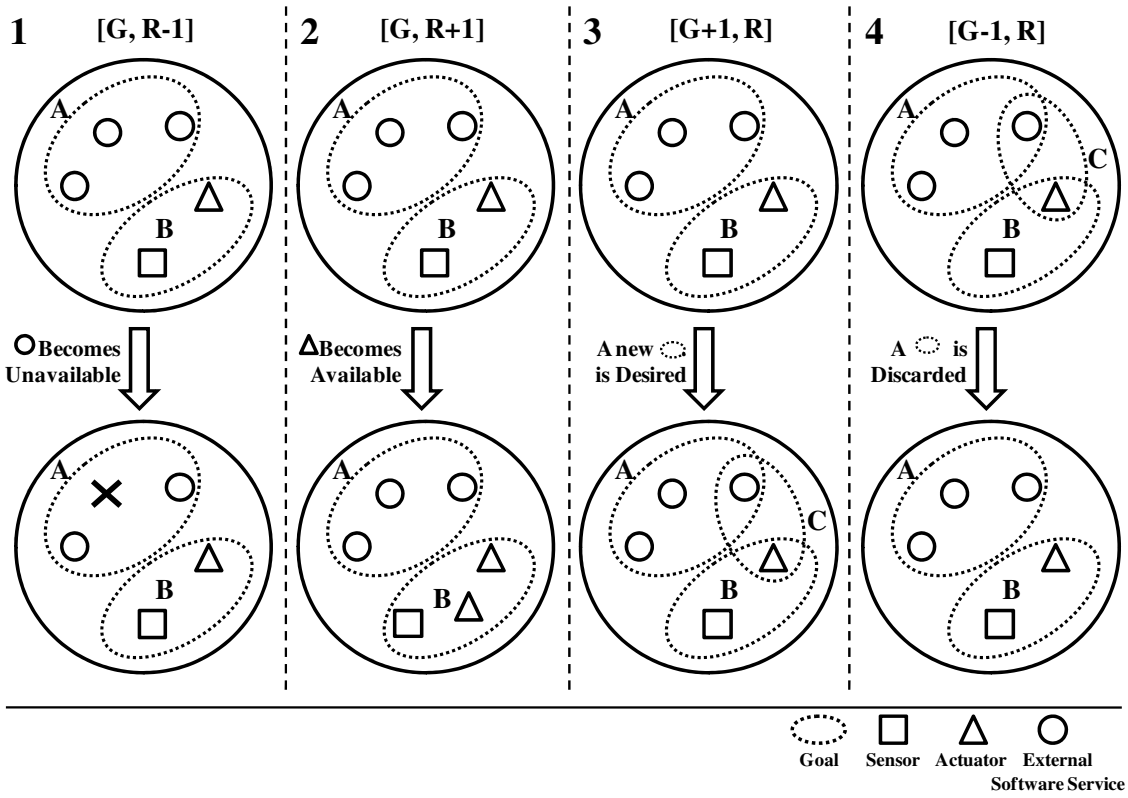

Figure 4.1: Evolution Scenarios

The aim of this categorization is to establish a correlation between the knowledge required for system evolution and the knowledge used to design the SPLs.

SPLs for pervasive systems accomplish the construction of software systems by evolving numerous resources: sensors, actuators and services provided by external software systems. These resources are meant to be used by the software system to achieve user goals [67]. The software system must dynamically adapt itself to achieve user goals according to the available pervasive resources without requiring participation from the user.

Since pervasive resources (R) are highly dynamic and user goals (G) can change over time, the following scenarios may arise:

1. **A resource becomes unavailable**. The software system must achieve the same goals using fewer resources, [G, R-1]. This scenario can be identified in domains such us automotive, mobile devices or urban domains. An example of this is the *Bosch Gasoline System* SPL for gasoline engine control units [68],

where the system needs knowledge about how to simulate an unavailable resource by using the available resources (see section 1, Figure 4.1).

2. **A new resource becomes available**. The software system can use a new resource to achieve the same user goals, [G, R+1]. This scenario can be identified in domains such us smart homes or mobile devices. An example of this is the SPL for home automation systems [69], where the system needs knowledge about how to involve the new resource to contribute to achieving the user goal (see section 2, Figure 4.1).

3. **The user pursues a new goal**. The software must achieve a new user goal with the same resources, [G+1, R]. This scenario can be identified in domains such us smart homes, health care or emergency situations. The user demands new functionality from the system that was foreseen when designing the SPL but that was not selected when producing the system. For instance, in a smart home when the users are ready to go on holidays, they may want the presence simulation functionality which deters thieves by acting as if there were people at home. (see section 3, Figure 4.1).

4. **The user discards a goal**. The software system must achieve fewer goals using the same resources, [G-1, R]. This scenario can be viewed as a particular case of scenario 2 (a new resource becomes available). Discarding a goal does not always imply eliminating its resources. The resources associated to the discarded goal are now available to support other goals (see section 4, Figure 4.1).

Through the analysis of different reconfiguration needs we have detected several suitable reconfiguration scenarios [70]. However, we have mainly classified them in two groups:

- Involution Scenarios: in these scenarios the set of active features changes but new features are not included. The new configuration of a system is performed using the available features, activating or deactivating them. This kind of scenario happens whenever an adaptation is triggered from the system.

- Evolution Scenarios: the set of features is modified (by the usage of a new feature or the update of an existing one). The new configuration of a system can be considered a new version.

These two kinds of scenarios can impact the system in a different degree. Involution scenarios involve only a change in the feature state (e.g., enable or disable a feature), so the possible combination of active features is limited. Thus, the number of involution scenarios is finite. However, evolution scenarios are difficult to be anticipated for a system as unknown features may appear.

Considering the time dimension, involution scenarios occur in a more unexpected way that evolution scenarios. Evolution scenarios commonly respond to an upgrade intent. These upgrading activities can be scheduled, making evolution scenarios more predictable (users can decide when to upgrade). However, involution scenarios might arise from unpredictable events such as a device breakdown or an unavailable service. In these cases, the fastest response is needed to restore an stable state of the system where functionality is at least maintained or reduced as few as possible.

4.3 SPLs for Dynamically Reconfigurable Systems

Although traditional SPL engineering recognizes that variation points are bound at different stages of development, and possibly also at runtime, it typically binds variation points before delivery of the software.

In contrast, DSPL engineers typically arent concerned with pre-runtime variation points. However, they recognize that in practice mixed approaches might be viable, where some variation points related to the environments static properties are bound before runtime and others related to the dynamic properties are bound at runtime.

Current DSPL approaches address the design of the dynamic reconfiguration in an ad hoc manner. There is a need for an approach that systematically models possible configurations of a system as a product family capable of automatically reconfiguring from one configuration of the family to another. This section describes an approach for dynamic software reconfiguration in software product families based on variability management. The solution presented in this chapter allows for the

automatic management of the evolution within the product family.

SPLs employ a two-life-cycle approach that separates domain and application engineering. Domain engineering involves analyzing the product line as a whole and producing any common (and reusable) variable parts.

Application engineering involves creating product-specific parts and integrating all aspects of individual products. Both life cycles can rely on fundamentally different processes for example, agile application engineering combined with plan-driven domain engineering.

To achieve the goal of dynamic reconfiguration, the system needs knowledge about itself and how to deal with the above scenarios. The *Scope, Commonality, and Variability* analysis (SCV) [71] that was made to design the SPLs (Domain engineering) can contribute to this dynamical reconfiguration. To reuse this SCV analysis, it is necessary to perform the following steps to transfer the knowledge from the SPL design to the run-time system in a systematic manner:

1. **Reusing the SPL Reconfiguration Knowledge**. Identify the useful knowledge from the SCV analysis to obtain the dynamic reconfiguration.

2. **Extend the SPL**. Extend the SPLs for pervasive systems to incorporate the relevant SCV knowledge to the pervasive system.

3. **Introduce the Autonomic Reconfigurator Component**. Improve the pervasive system architecture by using the SCV knowledge to obtain the dynamic reconfiguration.

The aim of these steps is to translate the relevant SCV knowledge from the SPL design to the SPL output by introducing an autonomic reconfigurator component.

4.3.1 Reusing the SPL Reconfiguration Knowledge

By following an MDD approach, the modeling languages gather the SCV knowledge, and the relevant knowledge for the scenarios can be obtained by performing model to model (M2M) transformations. Previous works [72, 73, 74] addresses this same issue of applying MDD to SPL development. Our work also uses MDD as a

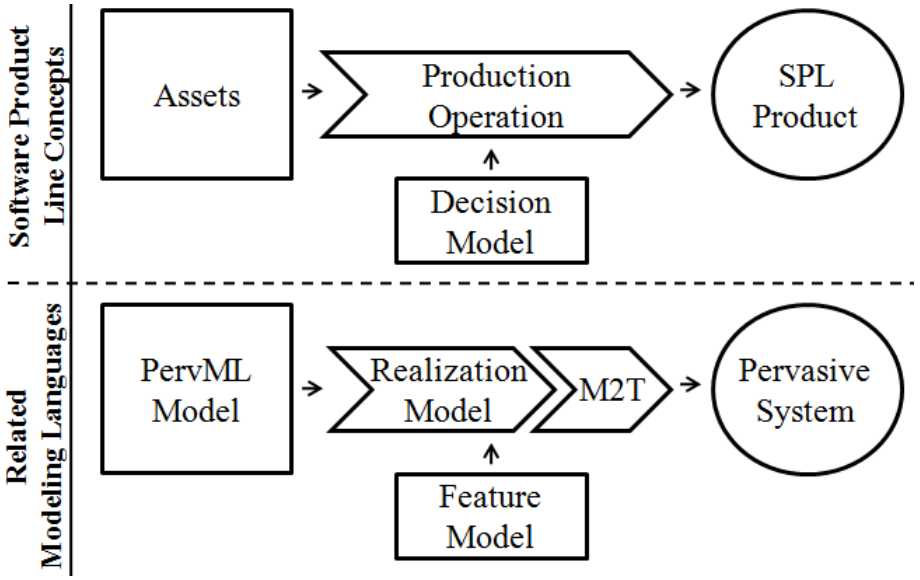

Figure 4.2: SPL following the MDD Approach

catalyst to transfer the knowledge from the SPL design to the SPL products. Figure 4.2 illustrates the relationship between these modeling languages and the SPL concepts. The following subsections present these modeling languages and identify the knowledge that can be reused to deal with the above evolution scenarios.

The PervML Model

Pervasive Modeling Language (PevML) [75] is a DSL for describing pervasive systems using high-level abstraction concepts. This language is focussed on specifying heterogeneous services in concrete physical environments such as the services of a smart home. These services can be combined to offer more complex functionality by means of interactions. These services can also start the interaction as a reaction to changes in the environment. The main concepts of PervML are: (1) a *Service* coordinates the interaction between suppliers to accomplish specific tasks (these suppliers can be hardware o software systems); (2) a *Binding provider* (BP) is a supplier adapter that embeds the issues of dealing with heterogeneous technologies; (3) an *Interaction* is a description of a set of ordered invocations between Services; and (4) a *Trigger* is an ECA rule (Event Condition Action) that describes how a Service reacts to changes in its environment. Figure 4.3 illustrates the relationships between these concepts. This DSL have been applied to develop solutions in the

smart home domain [76].

The PervML Model provides knowledge that is related to Scenario 2 (a new resource becomes available). The BP concept is the key element for managing new resources. The BP provides a level of indirection between the Services and the Resources. Resource operations interact with the environment (sensors and actuators) and provide functionality from external software systems. Services coordinate these resource operations to offer high-level functionality. If the resource operations do not match the Service expectations, then a BP is used to adapt these operations. Hence, the BPs decouple Services from resource operations. This property is essential for introducing extensibility to new devices and avoids having to modify existing Services to support the operations presented by new devices.

The FAMA Feature Model

Feature models are widely used to describe the set of products in a software product line in terms of features. In these models, features are hierarchically linked in a tree-like structure and are optionally connected by cross-tree constraints. There are many proposals about the type of the relationships and the graphical representation of feature models [77]. We have chosen the FAMA Feature Model [7] as the modeling language because it is oriented to feature reasoning and also because it has good tool support.

The FAMA Feature Model allows us to introduce knowledge related to Scenario 3 (the user pursues a new goal). From the point of view of the user, the goals represent his/her intentions [67]. From the point of view of the system, the features represent the functionality to support the user intentions. [78] presents mappings from user goals to feature models. To support a new user goal, the related features must be enabled. This step implies extending the production operation (see Section 4.3.2) to provide the required assets. The architecture must also be improved to introduce the knowledge from these models in order to dynamically integrate these assets. Thus, the goals are only limited by the available resources.

The Realization Model

The Realization Model is an extension that we have incorporated to Atlas Model Weaving (AMW) [79] to relate the SPL Features with the PervML elements. AMW is a model for establishing relationships between models. Our extension augments the *AMW relationship* with the *default* and *alternative* tags. This augmented relationship is applied between features and PervML elements (BPs and Services). In the context of a BP, the *default* relationship means that the BP is selected for the initial configuration of the system. The *alternative* relationship means that the BP is considered as a quiescent element that should be incorporated to the SPL product, but it does not participate in the initial configuration. Quiscent BPs provide an alternative BP to replace the default BP in case of fault. The more quiescent BPs that can be identified, the more flexible the adaptation will be.

The Realization Model provides knowledge related to scenario 1 (a resource becomes unavailable). The level of indirection introduced by the BPs facilitates the use of alternative resources, and the Realization Model determines the applicability of the BPs. Self-healing of the system is performed by applying a quiescent BP to a suitable resource. Quiescent elements and the Realization Model enable the Autonomic Reconfiguator to establish a dynamic binding at run-time.

4.3.2 Extending the SPL

Once the relevant SCV knowledge has been identified, it must be transferred to the SPL product. In this step, the Variability Model for the relevant SCV knowledge is obtained, and the model is prepared to be deployed in the software system platform. Figure 4.4 shows the extensions incorporated to the production operation. There are two new phases (Pruning and Wrapping) and the code generation phase has been augmented.

The Pruning Phase

A fundamental problem in SPL engineering is that a real product line can easily incorporate several thousands of variable features [80]. Incorporating variability

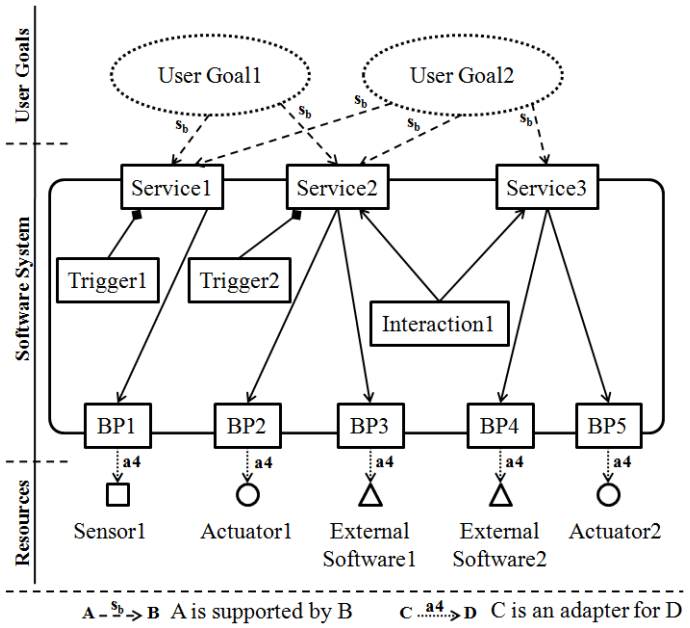

Figure 4.3: PervML Pervasive System

models to assist the system evolution impacts the complexity and system performance. The incorporated latency is determined by (a) *a model reasoner* and (b) *the size of the variability model*. The efficiency of the *model reasoner* is out of the scope of this work, and there are other works that address this problem [7]. The *size of the model* can be optimized by taking care of the specific evolutionary needs of each specific domain.

The *Pruning* phase performs model to model (M2M) transformations to prune the SCV models. For each one of the evolutionary scenarios that is not feasible (or interesting) in the specific domain, the pruning phase applies a set of pruning rules. These rules only prune the model elements that provide information that is related to the undesired scenarios. For example, the following algorithm describes the rules for pruning Scenario 3 (the user pursues a new goal):

1. Delete the relationships (from the Realization Model) that are reachable from an unselected feature (FAMA Feature Model).

2. Delete the model elements (from the PervML Model) that are not involved in a relationship (Realization Model).

3. Delete the features that are not selected by the user (FAMA Feature Model).

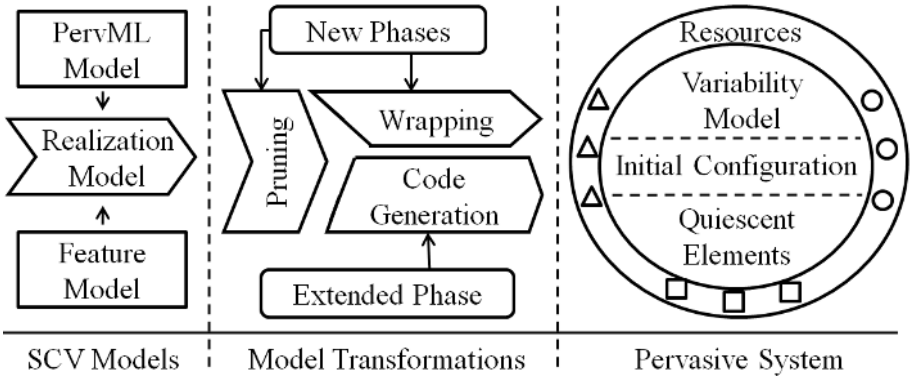

Figure 4.4: SPL Extensions

These rules have been implemented using the INRIA ATL [81] transformation language and tools. They take the SCV models as input and return a pruned varibility model. We have defined a set of ATL rules for each scenario, which can be applied in chain to prune more than one scenario.

The Wrapping Phase

The *Wrapping phase* is performed after the Pruning phase and has two steps. In the first step, the FAMA Feature Model, the Realization Model and the PervML Model are joined in a stand-alone XMI file (Variability Model). In the second step, the Variability Model is prepared for the specific software platform. This step depends on the deploying platform. In our case, the specific platform is OSGI [82] and the model has to be wrapped in an OSGI bundle.

In summary, we can state that the SCV Models describe a software system and its variants. Then, the pruning phase eliminates the variants related to the unfeasible (or not interesting) evolution scenarios, and the wrapping phase prepares the Variability Model for the deploying platform.

The Code Generation Phase

The *Code Generation phase* translates the model elements into the implementation code. This is performed by applying a model to text (M2T) transformation to a subset of the PervML Model. This subset is made up of the elements involved in a *default* relationship with a selected feature. The transformation takes the subset

model as input and generates the OSGI bundles with the Java implementation [83]. An OSGI bundle contains all the Java classes related to only one PervML concept (Service bundles, Interaction bundles, Trigger bundles and BP bundles). These bundles make up the initial configuration of the system and are installed and started in the OSGI environment.

The extension to this phase addresses the generation of the quiescent elements. These elements are those PervML elements that after the pruning phase are not related to any selected feature, or are related to an *alternative* relationship. The generated bundles represent variations to the system they are installed but are not started. These *alternative* bundles are the building blocks to perform the dynamic reconfiguration. The following chapter describes how the system is dynamically reconfigured using these bundles and the variability model.

4.3.3 Configurable Product Generation

The production of a Configurable Product (CP) differs a bit from common SPL product configuration, where features are selected or removed to produce a final product. A CP might be considered as a partial or staged configuration of a variability model where three kinds of feature are considered:

1. Discarded features: feature that are not deployed in a CP.
2. Active features: features that are deployed and activated in a CP.
3. Deactive features: features that are deployed in a CP but are not activated, however they are available for reconfiguration.

Our SPL use variability models to describe the derivable CPs. Each CP keeps the information about the dependencies and relationships among its features, using a CP-specific variability model. A CP variability model is generated from the SPL variability model where discarded features are removed. It is important to check both, the selection and the resultant CP variability model for containing no errors, such as mandatory features that have accidentally being removed or incompatible selections of features.

From a CP point of view, only two kinds of features are considered in variability models: active and deactive features. Reconfiguration is held on the reconfiguration component that is in charge of de/activating the features to adapt itself when adaptation triggers are received.

4.3.4 Decision Maker

A decision maker is in charge of reacting to the adaptation triggers, deciding which features must be activated or deactivated. The way to make decisions is not fixed and many alternatives may be considered such as rules system or *ad-hoc* reasoners that use logic paradigms. The information to be taken into account to make decisions is:

- The features available in the CP and their state (activated or deactivated)
- Dependencies among features.
- Adaptation triggers that inform about an involution scenario or a needed feature.
- User requests of features activation or deactivation.

Then, a variability model may be used to represent information relative to features and their relationships, and the adaptation trigger or user requests inform about a list of features to be activated or deactivated. In this kind of SPL, two decision makers are used, one at SPL side and another one at CP side. The responsibilities of a CP decision maker are:

- Asking the DSPL for new reconfigurations when an evolution or involution scenario is performed.
- Giving a response to evolution and involution scenarios whenever DSPL is not available.
- Communicating the reconfigurator for the features to be activated or deactivated in reconfigurations.
- Giving a fast response to involution scenarios by means of precalculated reconfigurations.

On the DSPL side, its decision maker is conscious of the whole structure of the DSPL by means of its variability model. Moreover, the SPL knows the big picture of the DSPL, its variability model manages more knowledge than a CP variability model and commonly pervasive systems are computationally more limited than other systems. Because of these two factors, we determine the following responsibilities of the DSPL decision maker:

- Calculating the reconfiguration in involution and evolution scenarios and send them to the CP configurators.
- Generating a CP variability model from the SPL variability model and the selected features.

FAMA Framework [84] is a SPL able to produce customized tools to reason about variability models. It uses different logic paradigms and algorithms to reason and extract information from variability models to help on decision making.

Although any solution may be given to implement a decision maker, we propose using FAMA Framework to generate both DSPL and CP decision makers. We specify the operations that are needed to extract information from a variability model and make decisions:

- Producing a CP variability model from the SPL variability model either when a CP is firstly produced or in an evolution scenario.
- Calculating involution scenarios by propagating decisions when a feature is de/activated.
- Providing explanations [85] when a reconfiguration is not possible.

As FAMA Framework is a SPL itself, we may create specific products that support above operations at both sides. This is important in CPs whose resources are limited and some functionality of FAMA Framework is never used. In this case, we are currently working on developing *FAMA Lite*, a specific product for hardware-limited CP.

4.3.5 Elaborating a Contingency Plan

Evolution scenarios are predictable and commonly come from an updating process and may be scheduled. A new CP is regenerated where existing features are maintained and new ones are optionally added. The DSPL decision maker is in charge of deciding about the new CP configuration so the CP decision maker has no responsibility on this process but receiving and storing the new active configuration.

However, involution scenarios might arise from unpredictable events such as a device breakdown or an unavailable service. In these cases, a the fastest response is needed to restore a stable state of the CP where functionality is at least maintained or reduced to a minimum.

The CP reconfigurator must be in charge of determining the features to be activated or deactivated to take the CP to a stable state. However, explaining a failure and restoring it [85] is an NP-hard problem that in many cases might not produce a fast response. Failure restoring problems may be solved by using heuristics that return a good response that may or may not be the best response. It will allow to give a fast response in a limited time at the risk of reducing the functionality of the CP more than it is needed due to a non-minimal diagnosis.

As the number of involution scenarios is finite and can be foreseen, a *contingency plan* may be built by the DSPL decision maker so the CP decision maker knows how to act in these cases, giving an optimum response and restoring the CP rapidly to the state where most of the services are correctly working.

When an involution scenario happens, the contingency plan must be regenerated. In order to generate it as fast as possible, the CP decision maker may ask the DSPL for producing part of the contingency plan or a complete one. Whenever DSPL decision maker is not available, the contingency plan may be calculated on CP decision maker idle times.

4.4 Conclusions

Current practices in adaptive systems development can not scale to produce highly complex systems in an effective and satisfactory way. SPL engineering approaches

can help to overcome this problem. Different proposals that are based on the SPL principles are available, all of them with strong and weak points (see Chapter 3 Related Work). In this chapter we have proposed an approach which integrates the SPL principles (in particular variability modeling) and the MDD principles.

In summary, this chapter has defined the overall approach to extend SPL for building autonomic Pervasive Systems. The approach focusses The approach focusses on reusing the SCV knowledge from the SPL design to the SPL products. We also described the adaptation scenarios where these pervasive systems can be applied. Next chapters introduce the adaptation strategy and the system architecture of these autonomic pervasive systems.

Chapter 5

The Reconfiguration Realization in both SPL and Products

5.1 Abstract

During the production process of a product in a SPL, a feature may be bound at different times: design time, compilation time, configuration time and runtime. Intensively using runtime binding, we would be able to produce reconfigurable products that may change their functionality even when they are deployed.

However, in our approach a product has the ability of changing the functionality automatically and autonomically, i.e. without the interaction of users. This grade of autonomy implies the use of techniques that provide the knowledge or reasoning ability to adapt a product at runtime.

In this chapter, we analyze the reconfiguration process both from the point of view of the SPL and the Products. Finally, we describe the key infrastructure components to support the reconfiguration process.

5.2 The Reconfiguration Strategy

In the environment full of embedded services that Pervasive Computing envisions there is little space for configuration at the user side. Users demand a minimal configuration effort to incorporate new features or repair their systems. Therefore,

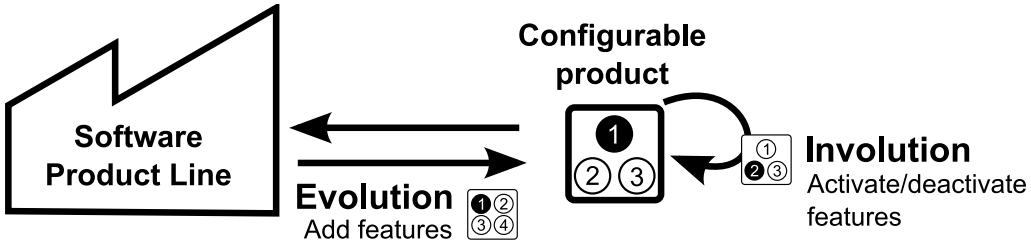

Figure 5.1: Involution and evolution scenarios

involution and evolution scenarios should be supported in a sound way. In this section, an strategy defined to cope with both kinds of scenarios is described.

Figure 5.1 illustrates an overview of this strategy for both kind of scenarios. Is worth noting that the Software Product Line is only involved in the evolution scenarios. While involution scenarios require no connection with the product line since the features included in the product are not augmented.

As the number of involution scenarios is finite and can be foreseen, a *contingency plan* may be built so that the system could know how to react in these cases, giving an optimum response and restoring itself quickly to the state where most of the services are correctly working.

5.2.1 The Reconfiguration Strategy from the Product Perspective

The *Autonomic Reconfiguration* component is responsible for applying the autonomic behavior to the system by performing dynamic bindings, taking the available resources and the Variability Model into account. This component is involved in the steps of resource discovery, model querying and reconfiguration execution. Figure 5.2 represents these steps graphically.

1. The user invokes a Service of the system to achieve a specific goal.

2. The software system core asks the Autonomic Reconfigurator Component for a dynamic binding of Services and Resources by using BPs.

3. The Autonomic Reconfigurator Layer discovers the available resources by using the discovery capabilities of the PervML framework [75].

4. The Variability Model is queried to performing the dynamic binding according to the available resources. The Autonomic Reconfigurator gets a set of relationships between Services and BPs from the Variability Model. If this set is empty, the reconfiguration procedure is stopped and the user is informed that there are no suitable resources to fulfill the goal.

5. The Autonomic Reconfigurator Component performs the binding of Services and BPs. If there is more than one feasible binding, the bindings that involve bundles from the *default* category are preferred. This step is performed by using the dynamic capabilities of the OSGI framework [86] to install, start, restart and uninstall Services and BPs (wrapped in OSGI bundles) without restarting the entire system.

6. Once all the bindings between Services and BPs are performed, the system is ready to access the available resources.

These steps are performed when the user starts the interaction using a service. However, in pervasive systems, the interaction can also be started by the resources. For instance, when a presence sensor detects a person in its action range, the alarm service is started. To cope with these situations, steps (3) to (5) are performed when there is a change in the availability of any resource.

5.2.2 The Reconfiguration Strategy from the SPL Perspective

We propose a self-reconfiguring strategy which addresses both evolution and involution scenarios. Figure 5.3 shows the strategy steps in order to perform reconfiguration by a pervasive system.

1. Both changes in the environment and in the system itself can trigger the adaptation process. These changes start the adaptation process.

2. The system calculates a new configuration to deal with the sensed change. These configurations are calculated using the variability models which describe the system in terms of features.

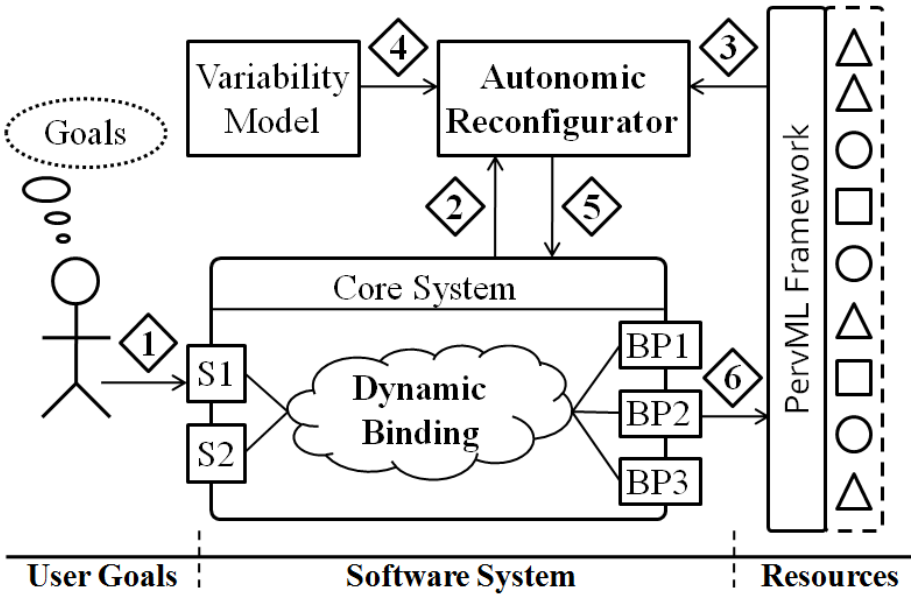

Figure 5.2: Reconfiguration Strategy

(a) If there is no configuration that resolves the adaptation trigger, then the system delegates the adaptation to the SPL. Therefore, the system sends information about the adaptation to the SPL. Optionally, the system can preprocess the information locally and send then a more specific information to the SPL.

(b) The SPL incorporates the acquired information to the product requirements and then it calculates a new system variant.

   i. If there is no variant that satisfies the product requirements, then the SPL notifies the system and the adaptation process fails.

(c) The SPL generates the update for the system. The update can be the whole calculated variant or the difference between the old variant and the new one.

(d) The SPL sends the update to the system.

(e) The system updates itself using the update from the SPL and the adaptation process ends.

3. The system reconfigures itself to apply the calculated configuration. The reconfiguration operation implies (1) starting/stoping components and (2) es-

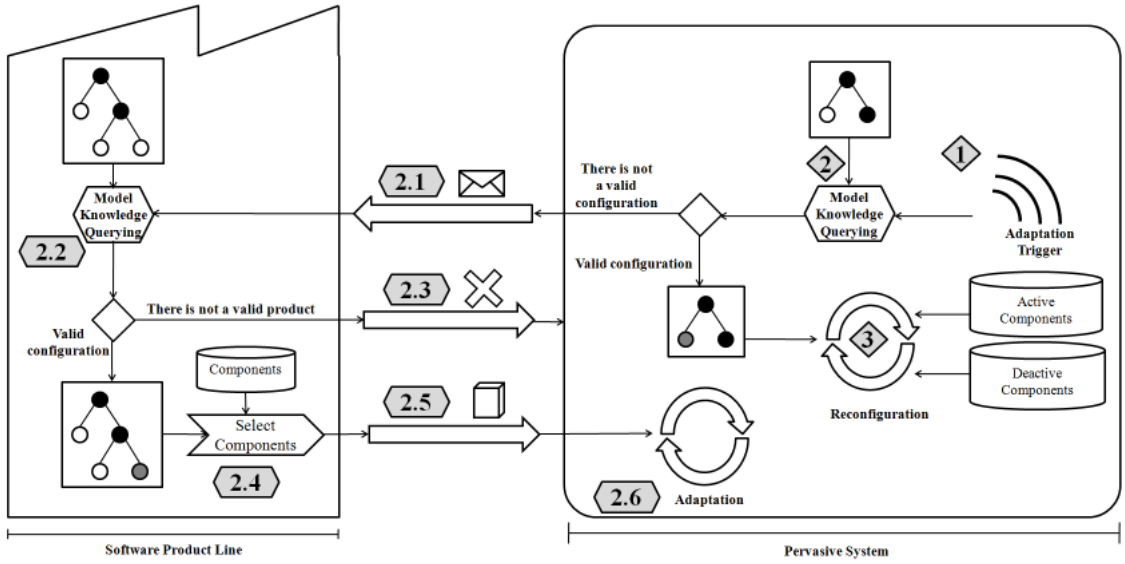

Figure 5.3: Adaptation Strategy

tablishing connections between them.

This is a global strategy which addresses both *evolution* and *involution* scenarios. Next, we analyze the strategy from either the evolution perspective or the the involution perspective. From the involution perspective, we characterize the strategy from the following points of view:

- **Autonomic degree.** The CP have no dependency of the SPL to perform the adaptation because there is necessary no connection between the SPL and the CP. The adaption depends only on CP resources.

- **Adaptation capabilities.** In general, the more variability knowledge the system has about itself and its variants, the more adaptable the system will get. This knowledge is captured in the variability models incorporated to the system. However, the variability models must be complemented with system components. Some components conform the initial system configuration, while the others are used in system reconfiguration. In conclusion, the adaptation capabilities depends on the knowledge captured in the models and on the number of components for system reconfiguration.

- **Computational overload.** The self-reconfiguring strategy introduces a computational overload to the system execution when the adaptation is triggered.

This overload comes from (1) the model queries and (2) the execution of the reconfiguration (starting stopping and linking system components).

From the evolution perspective, we characterize the strategy as follows:

- **Autonomic degree.** The system depends on the SPL availability to perform the adaptation, because the system gets **updates** from the SPL to perform the adaptation.
- **Adaptation capabilities.** To address an evolution scenario, variability models indicate the necessary components. However, some of these components are not in the system. In this case, the system has to get these components from the SPL. In this case, the adaptation capabilities depends on the available components at the SPL.
- **Computational overload.** Compared with involution scenarios, an evolution scenario introduces the following additional overload: 1) the communication with the SPL (to get system updates) and (2) the on-line installation of updates.

5.3 The Reconfiguration Framework

This reconfiguration strategy is supported by the variability models and a reconfiguration framework. The (1) reconfiguration framework provides the underlying infrastructure for adaptation. This framework is based on OSGI [82] and it implements a complete and dynamic component model where components can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot. Next sections describe the key infrastructure components of this framework.

5.3.1 Characterization Component

In the present work the context information is expressed by means of a list of the active contextualizers for each kind. We rely in XML namespaces to allow the definition of organization-specific contextualizers, enable a safe sharing and combination,

and the definition of standard vocabularies. The extensible capabilities of XML allow us to extend contextualizers with additional information. In this way, a more complex description of context information can be used. However, to take advantage of this the rest of the components should be updated to consider this specific information. The acquisition of the information in order to create this XML document can be performed by the user or it can be gathered automatically by means of sensors. For example, the user can explicitly state that she is starting a long journey. While, sensors can be used to detect her transition from an indoor environment to an outdoor one and adapt the service accordingly. In the developed prototypes we have considered the explicit elaboration of these documents since the acquisition of context information is out of the scope of the present work.

The active contextualizers are provided to the system in order to allow the adaptation of all their services to the context of use. Our implementation gathers this information from an XML document when the system is accessed. This information is processed to identify the known contextualizers and express them by means of key-value pairs as shown in Fig. .

5.3.2 Analyzer Component

The analyzer component actuates as a filter for the alternatives that are expressed in the Feature Model. For a given service, only the features that can fulfil the specified constraints are considered. The active contextualizers define a set of required, preferred, discouraged and forbidden properties. These requirements are used to decide which features are the most suited as it was illustrated in Fig. 3. In order to select a feature, this feature has to support the required properties and not support the forbidden ones. In addition, preferred features are favoured respect the disallowed ones. This selection operation is defined by means of the following expression in OCL:

```
Features.allInstances() ->
select(f|f.complete-> union(f.partial)->
containsAll(ctx.required)->
reject(f|f.complete->union(f.partial)->
```

```
containsAny(ctx.forbidden)->
sortedBy(f|ranking(ctx.preferred, ctx.discouraged, f))
```

The expression defines the set of available features given a certain set of contextualizers (ctx in the formula). First, features that support the required properties are selected; then, the ones that support forbidden requirements are excluded. The remaining features are ordered according to their preferred and disallowed properties by means of the ranking function.

5.3.3 Reconfigurator Component

Once the Analyzer component determines the suitable features, it is necessary to identify the resources that support these features and establish the resource-service bindings. This corresponds with the step 4 of the Execution Strategy. First, the Analyzer component provides the list of feature names which fulfil the contextualizer constraints, and then the reconfiguration component locates the references to the resources tagged with one of the feature names. Finally, the service-resource bindings are implemented using the OSGI Wire Class (an OSGI Wire is an implementation of the publish-subscribe pattern oriented to dynamic systems). The following Java method describes the actions to retrieve the reference of the selected resources:

```
public ServiceReference [] getServiceReference
(String filterString){
Filter filter = context.createFilter(filterString);
ServiceTracker tracker = new ServiceTracker(context,filter,null);
tracker.open();
ServiceReference [] selected; =
selected= tracker.getServiceReferences();
if (selected!=null)      return selected;
else return new ServiceReference [0];}
```

The above Java code uses the OSGI ServiceTracker to filter OSGI services (both PervML resources and PervML services are wrapped into OSGI services). The Service Tracker matches the services' properties against a filter parameter to narrow

down the query results. The OSGi filter is based on the LDAP search filter's string representation. For example, the filter to find the resources related with the In-Home-Detection feature is: "(Feature= In-Home-Detection)". Finally, the resources are bind with the service demanded by the user. The following Java method performs these bindings:

```
protected void createWires
(WireManager wm, ArrayList resourcesPIDs, String servicePID){
wm.deleteWires(consumerPID);
//Establish the new binding
String wireID = wm.createPID(producerPIDs, consumerPID);
Hashtable props = new Hashtable();
%props.put(PervML_WireID, wireID);
wm.createWires(producerPIDs, consumerPID, props); }
```

Before the new bindings are created, the old ones must be deleted. Previous resource-service bindings represent a previous dynamic customization of the services. The new bindings are created using OSGI Wire objects. A Wire object connects a resource (producer) to a service (consumer). Producer and Consumer services communicate with each other via Wire messages. Both producers and consumers implement OSGI interfaces (Consumer and Producer) to send/receive Wire messages. Once these wires are created, the system services have been customized. The services messages are communicated to the resources that better fulfil user contextualizers.

5.4 Conclusions

In this chapter, we analyze the reconfiguration process both from the point of view of the SPL and the Products. This reconfiguration process has been discussed by means of the following criteria: Autonomic degree, Adaptation capabilities and Computational overload. Finally, we described the the underlying infrastructure components to support the reconfiguration process: Characterization Component, Analyzer Component and Reconfigurator Component.

Chapter 6

Reconfiguration Architecture

6.1 Abstract

In previous chapters, a methodology based on SPLs principles was defined to cope with adaptivity of Pervasive Systems. This approach is based on the reuse of the knowledge from the design of SPLs to support adaptivity in the resulting systems. By means of model transformations, the SPL knowledge is systematically reused at run-time.

This chapter is focused on the model-based architecture for support some adaptation scenarios very common in Pervasive Systems (evolution and involution scenarios). These scenarios have different requirements regarding adaptation, and the way in which models are handled at run-time should consider those particular requirements. First, we discuses design patterns for System Adaptation and then the architecture to realize dynamic adaptation at runtime is presented. In this architecture different adaptation mechanisms can be offered depending on how much critical the adaptation is.

6.2 Architectural Design Patterns for System Adaptation

Software architectural patterns [87, 88] provide the skeleton or template for the overall software architecture or high-level design of an application. These include such widely used architectures [89] as client/server and layered architectures. Design patterns [90] address smaller reusable designs than architectural patterns, such as the structure of subsystems within a system. The description is in terms of communicating objects and classes customized to solve a general design problem in a particular context.

Basing a candidate software architecture on one or more software architectural patterns helps in designing the original architecture as well as evolving the architecture. This is because the adaptation and evolutionary properties of architectural patterns can also be studied and this assists with an architecture-centric evolution approach [91].

There are two main categories of software architectural patterns [92]. Architectural structure patterns address the static structure of the software architecture. Architectural communication patterns address the message communication among distributed components of the software architecture. Most software systems can be based on well understood overall software architectures. For example, the client/server software architecture is prevalent in many software applications. There is the basic client/server architecture, with one server and many clients. However, there are also many variations on this theme, such as multiple client / multiple server architectures and brokered client/server architectures. Furthermore, with a client/server pattern, the server can evolve by adding new services, which are discovered and invoked by clients. New clients can be added that discover services provided by one or more servers.

Many real-time systems [93] provide overall control of the environment by providing either centralized control, decentralized control, or hierarchical control. Each of these control approaches can be modeled using a software architectural pattern. In a centralized control pattern, there is one control component, which executes a

state machine. It receives sensor input from input components and controls the external environment via output components. In a centralized control pattern, evolution takes the form of adding or modifying input and/or output components that interact with the control component, which executes a state machine. Another architectural pattern that is worth considering because of its desirable properties is the layered architecture. A layered architectural pattern allows for ease of extension and contraction [94] because components can be added to or removed from higher layers, which use the services provided by components at lower layers of the architecture.

In addition to the above architectural structure patterns, certain architectural communication patterns [88] also encourage adaptation and evolution. In software architectures, it is often desirable to decouple components. The Broker, Discovery, and Subscription/Notification patterns encourage such decoupling. With the broker patterns, servers register with brokers, and clients can then discover new servers. Thus a software system can evolve with the addition of new clients and servers. A new version of a server can replace an older version and register itself with the broker. Clients communicating via the broker would automatically be connected to the new version of the server. The Subscription/Notification pattern also decouples the original sender of the message from the recipients of the message.

The software architecture is composed of distributed software architectural patterns, such as client/server, master/slave, and distributed control patterns, which describe the software components that constitute the pattern and their interconnections. For each of these architectural patterns, there is a corresponding software adaptation pattern, which models how the software components and interconnections can be changed under predefined circumstances, such as replacing one client with another in a client/server pattern, inserting a control component between two other control components in a distributed control pattern, etc.

A software adaptation pattern defines how a set of components that make up an architecture or design pattern dynamically cooperate to change the software configuration to a new configuration given a set of reconfiguration commands. A software adaptation pattern requires state- and scenario-based reconfiguration be-

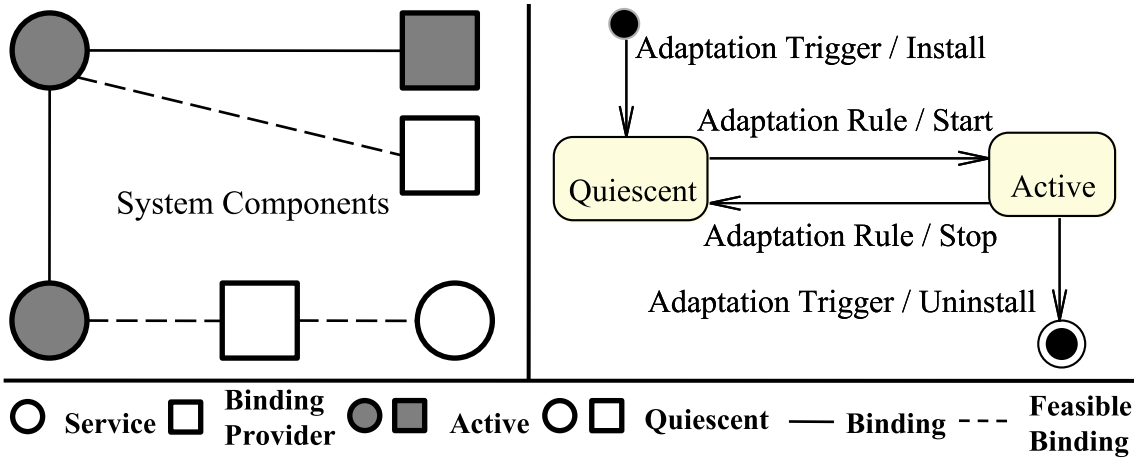

Figure 6.1: System components.

havior models to provide for a systematic design approach. The adaptation patterns are described in UML with adaptation integration models (using communication or sequence diagrams) and adaptation state machine models [95, 96]. An adaptation state machine defines the sequence of states a component goes through during from a normal operational state to a quiescent state, as shown in Figure 1. Once quiescent, the component is idle and can be removed from the configuration, so that it can be replaced with a different version of the component.

6.3 The Adaptive Architecture

To perform adaptation, our approach is based upon a framework for adaptive systems proposed in [97] by analyzing common terminology and synergy between different approaches. This framework introduces the roles of (1) triggers which specify the event or condition that causes the need of adaptation; (2) adaptation actions which realize the actual adaptation; and (3) adaptation rules that define which triggers cause which adaptation actions. In our approach, these rules are driven by run-time models to modify the system architecture using adaptation actions.

6.3.1 The Underlying Components

In order to allow a flexible adaptation process, we have considered an architecture based on communication channels (called bindings). This architecture for the final

system allows an easy reconfiguration process since communication channels can be established dynamically between the components that form the system (see left of Fig. 6.1). These components are classified in Service and Binding Providers as follows:

- **Service.** A *Service* coordinates the interaction between resources to accomplish specific tasks (these resources can be hardware or software systems);
- **Binding Provider.** A *Binding provider* (BP) is a resource adapter that handles the issues of dealing with heterogeneous technologies. The BP provides a level of indirection between Services and resources. Resource operations interact with the environment (sensors and actuators) and provide functionality from external software systems. Services coordinate these resource operations to offer high-level functionality. If the resource operations do not match the Service expectations, then a BP is used to adapt these operations. Hence, the BPs decouple Services from resource operations.

For example, in a smart home a security service is composed of several resources such as presence sensors, movement detectors, sirens, contact detectors, SMS senders, silent alarms and so on. The security service coordinates the behaviour of all these resources.

6.3.2 Adaptation Actions

The system architecture has to be modified as a result of the dynamic adaptation. Old components must be dynamically replaced by new components while the system is executing. The adaptation actions are in charge of this dynamic reconfiguration. These actions deal directly with the system components by means of the following operations: Component State-Shift and Component Binding.

1. **Component State-Shift** Kramer and Magee [98, 99] described how in an adaptive system, a component needs to transit from an active (operational state) to a quiescent (idle) state in order to perform the system adaptation.

We have applied this approach to our systems by means of the OSGI framework [82]. The OSGI Framework defines a component life cycle where components can be dynamically installed, started, stopped, and uninstalled (see right of Fig. 6.1). On the one hand, *Triggers* are in charge of perform the install/uninstall operations. For example, when a resource fails or a new resource is installed in the system. On the other hand, *Adaptation Rules* are in charge of perform the start/stop operations. For example, when a Binding Provider must be activated to handle a new resource.

2. **Component Binding** Once a component transits to an active state, it needs bindings with other components. These bindings are implemented by using the OSGI Wire Class (an OSGI Wire is an implementation of the publish-subscribe pattern oriented to dynamic systems). The OSGI Wires establish communication channels between components to send messages one another.

Adaptation actions provide the basics operations to dynamically change the system architecture. Adaptation rules orchestrate the execution of these actions by means of the run-time models. The next section details how the adaptations rules queries the models in order to apply the adaptation actions.

6.4 Adaptation Rules

In a nutshell, an adaptation rule is in charge of (A) handling the adaptation triggers, (B) gathering the necessary knowledge from the run-time models and (C) applying the adaptation actions.

As we state above, evolution and involution scenarios present different requirements. In involution scenarios the system must provide an autonomic response in a reduced amount of time. While in evolution scenarios, the system does not present the same time requirement and even the user might assist the adaptation. To fulfil theses requirements, we have defined two kinds of adaptation rules taking into account the type of scenario.

6.4.1 Adaptation in Evolution Scenario

When a component is plugged-in, first the adaptation rule queries the feature model for which new features could potentially be activated. Then the user confirms the features activation. Furthermore, activating new features can fulfil other feature constraints which might be enabled. Therefore, each time the user confirms a feature activation, the adaptation rule queries the feature model for new features. Finally, the Component and Structure Models drive the adaptation actions in order to dynamically reconfigure the system architecture and support the new features. The steps to perform this adaptation (see Fig. 6.2) are detailed next:

1. By means of the Component model, the adaptation rule identifies those features which are related to the trigger component. With these features, the rule creates an ordered set called the *evolution set*. For each one of the features, the rule performs the following steps, 2 to 5.

2. The rule checks the possibility of feature activation. This information is in the Feature Model, specifically it depends on the requires, excludes and mandatory relationships between features. If all these constraints are fulfilled, then the feature can be activated.

3. Once the rule checks the feature activation, it asks the user for confirmation by means of a dialog in the user interface. The message shows the name of the feature and a description stored in he Feature Model. The message also provides three options to the user: “Yes”, “Remind me later” and “No”.

4. Activating a new feature can fulfil other feature constraints. In this step, the rule checks for new activable features. The rule adds these new features to the *evolution set*.

5. In terms of the platform, activating a feature implies performing *adaptation actions* to system components. In this step, the rule queries the Component model for the feature components. For each one of these components, the rule performs the following steps, 6 and 7.

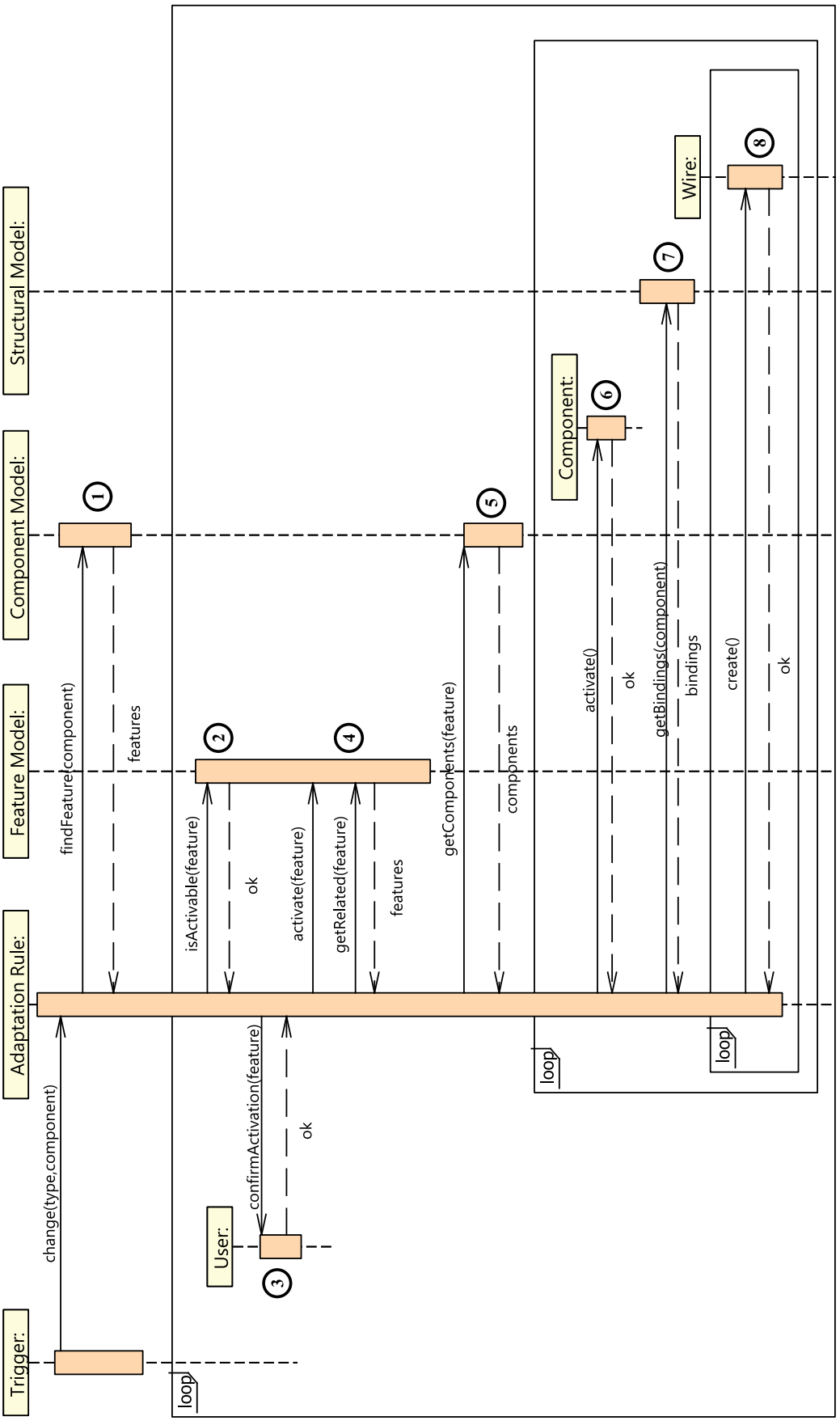

Figure 6.2: Adaptation process for evolution scenarios.

6. The rule applies the *State-Shift* action to the component. Therefore, the component transits from a quiescent state to an active state.

7. To connect the new active component with the rest of the system, the rule queries the Structural model for the component bindings.

8. Finally, the rule applies the *Binding* action to create the communication channels between the components.

Due to space constraints, the sequence diagrams in this section represent only the general case for adaptation. Diagrams consider only affirmative responses, lacking alternative behaviour.

In our experience applying this approach to the smart home domain [70], we have notice that the time response delay comes mainly from these factors: feature dependency resolution (steps 2 and 4) and user confirmation (step 3). How much time the user takes to confirm can not be foreseen, and dependency resolution is more time consuming than other simpler queries (for example, step 7). However, we consider that installing new resources in the system is not as critical as handling resource failures. Thus, in evolution scenarios we offer an advance system response (dependency resolution and user participation) although this response takes extra time.

6.4.2 Adaptation in Involution Scenario

Involution scenarios are triggered by the removal of a resource. A fast adaptation of the system is required to minimize the impact of the lost resource. In order to offer a good response time, adaptation is automatic (not requiring user intervention) and resource alternatives are precalculated in a model (the realization model). In this way, the latency of asking the user is avoided and the effort of reasoning with the feature model (e.g., looking for dependencies) is also reduced.

In Fig. 6.3, the adaptation process for an involution scenario is illustrated. Given the removal of a component, the affected feature is obtained and an alternative component for this feature can be directly retrieved from the Realization Model. More detail about the process is given below:

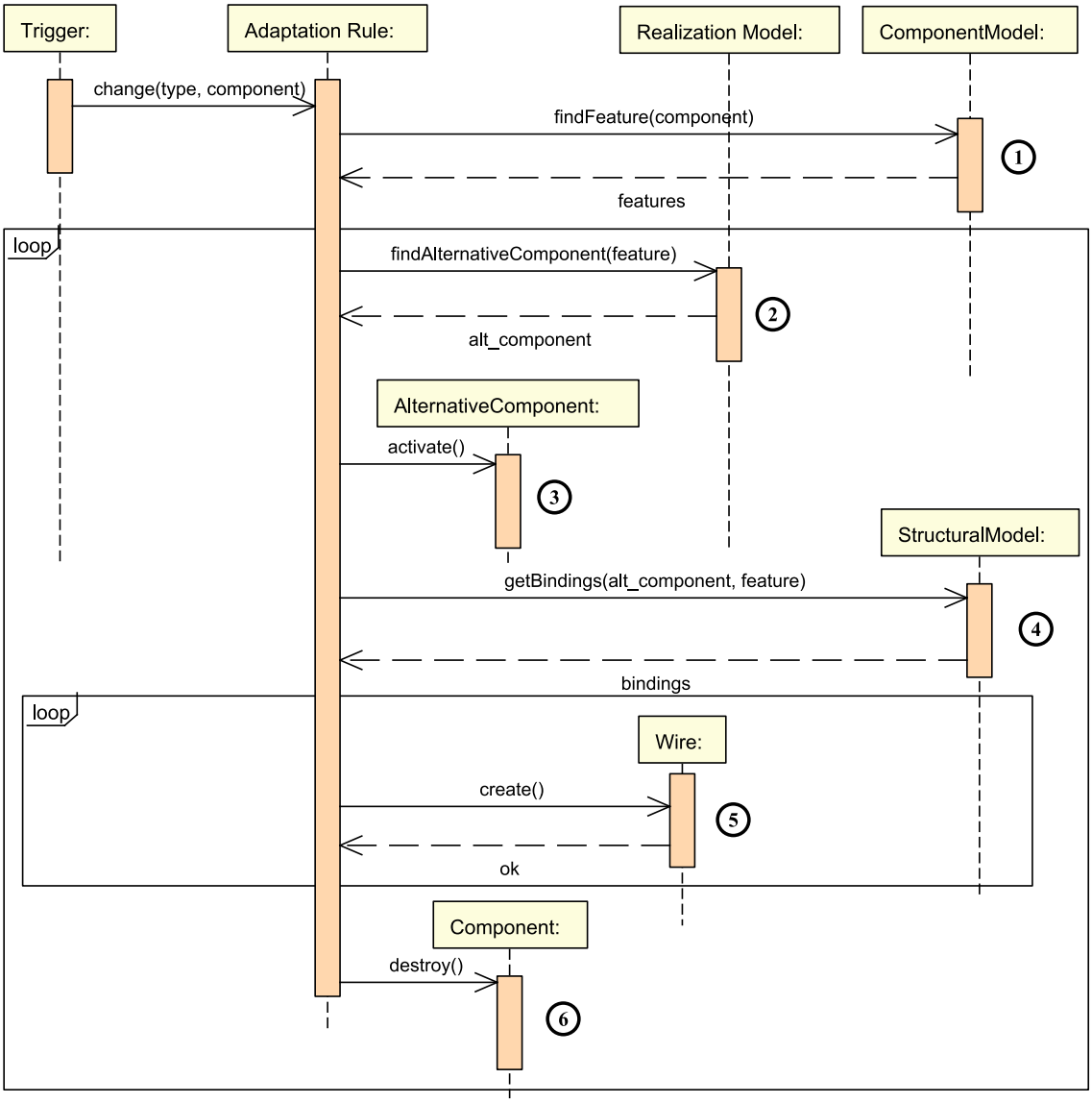

Figure 6.3: Adaptation process for involution scenarios.

1. When a change is produced in the system, the affected features are obtained in the same way as in the evolution scenario. The following steps are performed for each feature.

2. The rule queries the Realization model to obtain a component that can replace the affected one for a given feature. Since this information is expressed explicitly in this model, queries are straightforward.

3. Once the rule has found an alternative component (initially in the quiescent state) it is activated.

4. The alternative component may require communication with other components. This information is obtained from the Structural model.

5. For each of the required bindings, a wire is created to establish the necessary communication channel between components.

6. Finally, the affected component is destroyed. This implies the removal of inactive wires. The destruction of this component is deferred until the end of the adaptation process, since the priority in involution scenarios is to offer the new services immediately.

The adaptation rule for involution reduces the delays commented for the evolution scenarios. On the one hand, model queries are simplified. Reasoning over a feature model is a time-consuming activity and termination becomes difficult to guarantee [100]. On the other hand, the user does not participate in the process, which is a requirement for the autonomic behavior required by this kind of scenarios.

6.5 Conclusions

In this chapter, we provide support for adaptation in pervasive systems by means of run-time models. Our approach focusses on addressing the differences between evolution (a resource is added) and involution (a resource is removed) scenarios. In involution scenarios, we use models with precalculated knowledge in order to provide an autonomic response in a reduced amount of time. While in evolution

scenarios, we offer an advanced system response (feature dependency resolution and user participation) because we consider that installing new resources in the system is not as critical as handling resource failures. Finally, we showed how models drive the system adaptation within the context of each scenario.

Chapter 7

Case Study:
An Autonomic Smart Home

7.1 Abstract

We illustrate the proposed SPL for Autonomic Pervasive Systems by modeling a smart home family: a localized technology-augmented environment where people perform everyday life activities. This chapter presents the models of the SPL and the output smart home produced by the SPL. Then, we identify a set of adaptation-scenarios to check the adaptation capabilities. Finally, we describe how the smart home reconfigures itself when the adaptation-scenarios are applied.

7.1.1 The Smart Home Family Description

The SPL developed for this case study addresses automated lighting, presence detection and security functionality for the smart homes. The SPL models describe the collection of all smart homes that can be produced. A smart home is uniquely defined by the selections on the Feature Model. These selected features determine (by means of the Realization Model) which elements of the PervML Model are used for the initial configuration of the smart home.

From an adaptative point of view, the unselected features of the feature model represent variants to the selected smart home. These unselected features determine

which elements of the PervML Model are used to dynamically reconfigure the system. All these models are presented as follows:

1. **The FAMA Feature Model**. This model (see the top of Figure 7.1) determines the initial and the potential features of the smart home. The grey features are the features selected to specify a member of the smart home family. The white features represent potential variants. Initially, the smart home provides automated lighting and a security system. This security system relies on perimeter presence detection (outside the home) and a visual alarm. The system can potentially be upgraded with in-home presence detection and more alarms to enhance home security.

   As stated in section 4.3.1, supporting a new user goal is translated into enabling more features. Some features can be enabled by plugging in new physical resources, while other features can be enabled because the restrictions (*mandatory*, *excludes* or *requires*) are resolved. For instance, features (8) and (13) can be enabled if a volumetric detector is plugged in. Then, the *requires* dependency from (2) to (8) can be resolved.

2. **The PervML Model**. This model (see the bottom of Figure 7.1) describes the building blocks for the assembly of a pervasive system [75]. The grey blocks implement the functionality of the selected features. The white blocks enable the reconfiguration of the system. The (a) and (e) blocks implement the functionality for the unselected Presence Simulation feature. The (i), (k), (l), (m), and (o) blocks provide adapters for the new resources that can become available, as mentioned in Section 4.3.1. The Autonomic Reconfigurator prioritizes the grey blocks over the white blocks since they are related to the original features of the system, as stated in Chapter 4.

3. **The Realization Model**. This model (see the middle of Figure 7.1) establishes the relationships between the features and the PervML elements. Section 4.3.1 introduced the *alternative* relationship in order to identify BPs and Services that mitigate system faults. For instance, the visual alarm feature is

related to a BP (n) for visual alarms, but, alternatively, it can be replaced with a BP (k) that emulates the visual alarm by using the general lighting.

7.2 Adaptation-Scenarios

The adaptation-scenarios are designed to evaluate the adaptation capabilities of the smart home. They describe changes in the physical environment or in the user intentions according to the scenarios described in Chapter 4. These adaptation-scenarios are performed using our testbed (see Figure 7.2), which features a scaled smart home driven by a barebone-gateway, an Ultra Mobile PC (UMPC) for displaying the user interfaces, and several European InstaBus (EIB) devices. This smart home represents the physical resources for performing hot plugging tests. The barebone runs the OSGI server where all the Services, BPs, triggers and interactions are deployed, and it also runs the non-physical resources such as a weather-forecaster or an instant messaging client. Figure 7.3 shows the EIB devices related to the adaptation-tests.

1. **A new resource becomes available**. The security system relies on a presence detection service. This service integrates the functionality of several sensors. Hence, the more coverture the sensors have, the more reliable the service will be. In this adaptation-scenario we improve the coverture by incorporating a new in-home volumetric sensor (see the top of Figure 7.3).

2. **A resource becomes unavailable**. In a security system, the alarm is a key element. A fault (or manipulation) of this resource can invalidate the entire security system. The aim of this resource is to alert the neighbors of an unexpected situation in the house. There are several kinds of alarms such as visual, acoustic, or silent alarms. This adaptation-scenario focusses on dynamically replacing a damaged visual alarm with another one that consists of a fast and constant blinking of all the lights in the home (see the middle of Figure 7.3).

3. **The user pursues a new goal**. The addition of new resources to the system reinforces the support to current goals. It can also potentially enable the sys-

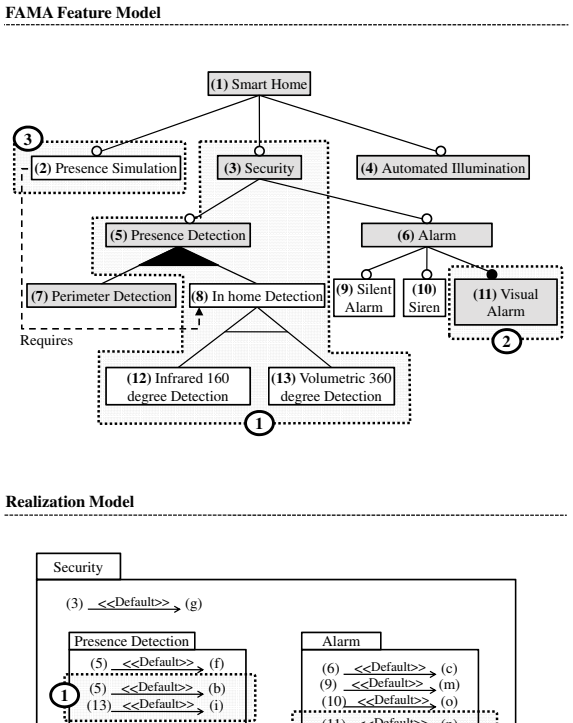

Figure 7.1: Models for the SPL

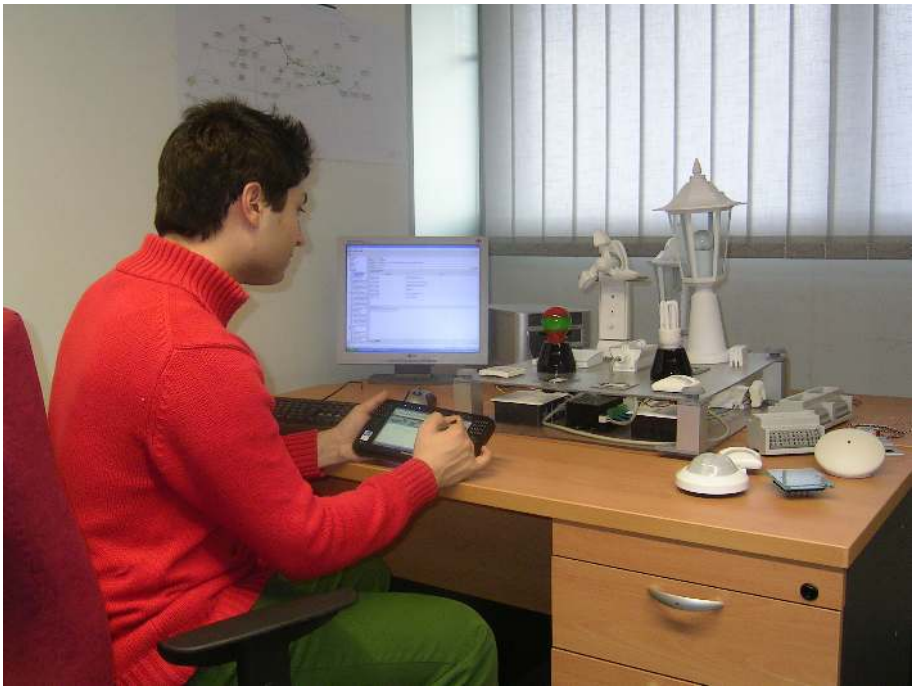

Figure 7.2: Testbed

tem to support new goals. Achieving presence simulation involves automated lighting and an in-home presence sensor (see Figure 7.1). The new plugged in-home sensor enables the system to offer the presence simulation service, which was not previously supported (see the bottom of Figure 7.3).

7.2.1 Smart Home Reconfiguration

The knowledge from the SPL models enables the Autonomic Reconfigurator to establish a dynamic binding between the system components. When the adaptation-scenarios are applied, the system reconfigures itself as follows:

1. **A new resource becomes available**. When a new resource is discovered by the PervML Framework a connection with the related Services must be established. This corresponds to steps 3 to 5 of Chapter 5. First (step 3, Figure 5.2), the PervML Framework identifies the *category ID* of the new resource for the Autonomic Reconfigurator. To identify the BP that is appropriate for this resource (step 4, Figure 5.2), the Autonomic Reconfigurator queries the PervML Model with the *category ID*. The Autonomic Reconfigurator must

also identify the Services that have to be aware of the BP. The Realization Model is queried for the features related to the BP. Then, the Autonomic Reconfigurator navigates from these features to the first parent feature that is related to a Service. The set of parent features indicates the Services that have to be used. Finally, the dynamic binding between the BP and these Services is performed (step 5, Figure 5.2). These bindings are implemented by using the OSGI Wire Class (an OSGI Wire is an implementation of the publish-subscribe pattern oriented to dynamic systems). The BP for the new resource (in a quiescent state) is started and subscribed to the identified Service wires.

This behavior was tested by applying the first adaptation-scenario of section 1 (see the top of Figure 7.3). When the Volumetric Detector is plugged, the Presence Detection Service needs to be aware of its notifications. The Volumetric Detector BP is related to feature (13), and the first parent feature that is related to a Service is the one labeled with a (5). Hence, the Volumetric Detector BP is subscribed to the wire of the Presence Detected Service. The relevant model elements of this example are denoted with the number 1 in Figure 7.1.

2. **A resource becomes unavailable**. When the PervML framework detects an unavailable resource, a dynamic binding to an alternative BP is requested from the Autonomic Reconfigurator. Steps 2 to 5 of Chapter 5 are applied to perform this binding. First (step 2, Figure 5.2), the active Service asks the Autonomic Reconfigurator for a dynamic binding with the resources. The PervML Framework looks for the available resources (step 3, 5.2). The autonomic framework queries the models (setp 4, 5.2) for the feature that represents the active Service of step 2. The BPs that are related to this feature (Realization Model) are evaluated against the available resources. If there is no BP available with the *default* tag, then the first BP with the *alternative* tag is selected. The binding is performed by susbscribing the selected BP to the wire of the active Service (step 5, 5.2).

   This behavior is tested by applying the second adaptation-scenario of Section

1 (see the middle of Figure 7.3). When the Visual Alarm fails, the Alarm Service needs an alternative supplier. The Automated Lights can be adapted to simulate a visual alarm. The BP (k) is selected to adapt the behavior of the Automated Lights by performing a continuous blinking. The relevant model elements of this example are denoted with the number 2 in Figure 7.1.

3. **The user pursues a new goal**. Plugging in new resources can enable the system to support new user goals. The features that are related to the new resources can resolve the restrictions (*mandatory*, *excludes* or *requires*) of other features. However, reasoning about feature dependencies is not a trivial task. We have implemented a toy reasoner within the Autonomic Reconfigurator using the EMF runtime. This is just a proof of concept, we plan to integrate the FAMA reasoner based on constraint programming. The reasoner determines the new features that can be enabled. Then, the Autonomic Reconfigurator starts the quiescent bundles and performs the suitable subscriptions to the OSGI wires.

   This behavior is tested by applying the third adaptation-scenario of Section 1 (see at the bottom of Figure 7.3). The new Volumetric Detector involves features (13) and (8), and feature (8) resolves the *require* dependency of feature (2). The relevant model elements of this example are denoted with the number 3 in Figure 7.1.

7.3 Conclusions

When end-users want to accomplish a particular activity they might use numerous devices and services in the process. The lines between these two are blurred, and users may not clearly distinguish between them. Hence, when something goes wrong (or right) it may be hard to correctly attribute it to the service or the device. Consider the case of a *Smart Home Security* service. This service relies on heterogeneous sensors. When a sensor goes down and users are not notified of unexpected presence in the some, their perceive that the whole service is malfunctioning.

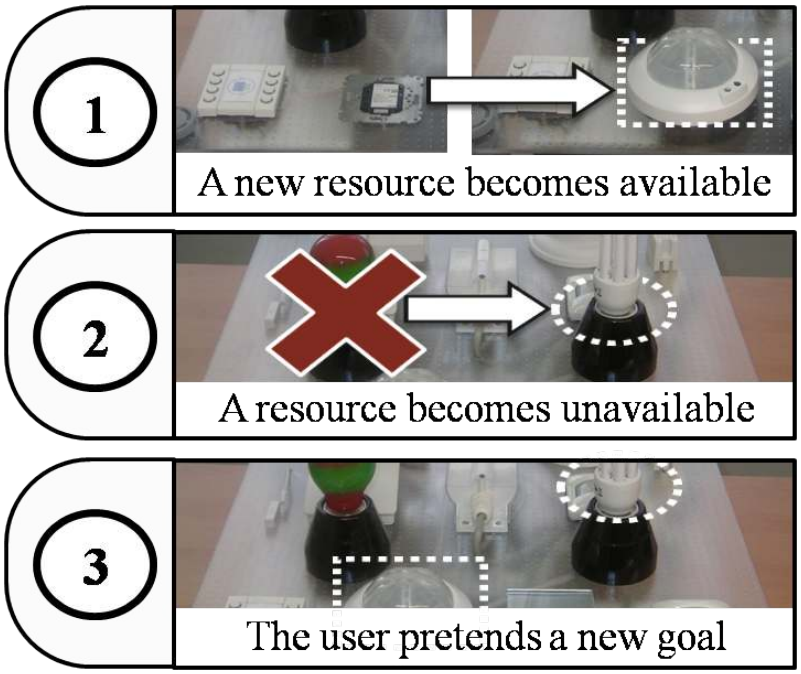

Figure 7.3: Adaptation Tests

The seamless integration of services and devices presents a challenge to the upgrading of smart homes. In some cases, it take too long to find the appropriate feature to tune the behavior of services, or it can not be find at all. This suggests that we need advanced mechanisms to manipulate feature models. We think that the following mechanisms can improve the manipulation of feature models:

1. **Filters**. A filter acts as a limitation for the potential configurations. A typical application of this mechanism occurs when end-users are looking for a configuration with a specific set of features, that is, they are not interested in all the potential configurations, but only in some of them (those configurations that pass the filter).

2. **Optimum configurations**. The Optimum configurations mechanism acts by finding the best configurations according to a specific criterion. Most end-users suggested criteria related to economical factors, such as asking for a configuration with the security service and the minimum price.

Chapter 8

Conclusions

8.1 Abstract

In this Master Thesis, we have proposed a model-driven Software Product Line (SPL) for developing autonomic pervasive systems. The work focusses on reusing the Variability knowledge from the SPL design to the SPL products. This Variability knowledge enables SPL products to deal with adaptation scenarios (evolution and involution) in an autonomic way. We have presented SPL extensions to transfer this knowledge to the SPL products. Finally, the product architecture has been improved in in order to be able to reuse this knowledge.

This chapter reviews our central results and primary contributions, evaluates the limitations of this work, and proposes new areas for future research.

8.2 Results and Contributions

We have stated that the work of this thesis is mainly related to two engineering paradigms: Software Product Lines (SPL) and Autonomic Computing (AC). The work of this thesis is related to SPL because we have presented a new SPL approach for producing autonomic pervasive systems. The work of this thesis is related to AC because this approach uses variability at run-time in order to dynamically self-reconfigure pervasive systems without user intervention.

In this context, and having a more detailed look to the work of this thesis, we

can state that the main contributions that we have introduced are the following:

1. In Chapter 3 we elaborated **a taxonomy of SPLs for adaptive products**. We intend to summarize the SPL architectures that have been proposed at date, dividing them in connected and disconnected SPL depending on their dependence with the SPL infrastructure. To bridge the gap between connected and disconnected SPLs, we have proposed an hybrid approach called mixed SPL. In Mixed SPLs some level of adaptivity is guaranteed, even if the SPL-Product connection is unavailable. Hence, the SPL-Product connection is not strictly necessary to achieve some level of adaptation. Although, SPL-Product connection is necessary to achieve fully adaptivity.

2. Chapter 4 proposes a model-driven SPL for developing autonomic pervasive systems. The main contribution is that **this process systematically reuses the variability knowledge from the SPL design to the SPL products**. This variability knowledge enables SPL products to deal with evolution in an autonomic way. We have described suitable scenarios where this knowledge can be applied. We have presented SPL extensions to transfer this knowledge to the SPL products. Finally, we have made an improvement in architecture (Autonomic Reconfigurator) in order to be able to reuse this knowledge at run-time. With this SPL knowledge, the resulting pervasive system can perform dynamic bindings to reconfigure itself without being connected to the SPL. We have successfully applied our approach to adaptive pervasive systems for smart homes, which are based on the scenarios described in [76].

3. Chapter 5 defines a strategy for the reconfiguration of pervasive systems at run-time based on solid engineering approaches of both fields, pervasive systems and SPL. PervML and FAMA feature model **allowed us to work at modelling level providing a high level perspective of pervasive system families**. Thanks to these techniques we can capture the variability of Pervasive Systems by means of models. These models can be later used by the system to decide how to adapt itself in both evolution and involution scenarios. In this way, when some resources are incorporated or missing from the

system, the system can keep providing its services in the best available way.

4. In Chapter 6, we provide support for adaptation in pervasive systems by means of run-time models. Our approach focusses on **addressing the differences between evolution (a resource is added) and involution (a resource is removed) scenarios**. In involution scenarios, we use models with pre-calculated knowledge in order to provide an autonomic response in a reduced amount of time. While in evolution scenarios, we offer an advanced system response (feature dependency resolution and user participation) because we consider that installing new resources in the system is not as critical as handling resource failures. Finally, we showed how models drive the system adaptation within the context of each scenario.

5. In Appendix A. We propose some **visualization techniques to make Feature Models editors scale as the model grows**, considering also user customization. We have analyzed the particular visualization needs demanded by Feature Modeling to detect some points of improvement. Finally we have developed Moskitt Feature Modeler (MFM), a graphical editor that addresses the detected limitations offering editing capabilities suitable for the feature modeling of large systems. In addition, MFM support the tri-state configuration of feature model introduced in this master thesis.

8.2.1 Publications

Parts of the results presented in this thesis have been presented and discussed before on distinct peer-review forums. The distinct publications in which the author of this thesis was involved are listed below.

1. **Carlos Cetina**, Joan Fons & Vicente Pelechano. Applying software product lines to build autonomic pervasive systems. 12th International Software Product Lines Conference (SPLC). Limerick, Ireland. 2008.

2. **Carlos Cetina**, Pau Giner, Joan Fons & Vicente Pelechano. A Model-Driven Approach for Developing Self-Adaptive Pervasive Systems. Models at run.time

08 Workshop in conjunction with MODELS (Models@run-time). Touluse, France. 2008.

3. Pau Giner, **Carlos Cetina**, Joan Fons & Vicente Pelechano. Adaptivity in Ubicomp Systems: dealing with different services and interaction mechanisms. 3rd Symposium of Ubiquitous Computing and Ambient Intelligence (UCAMI). Salamanca, Spain. 2008.

4. **Carlos Cetina**, Pablo Trinidad, Vicente Pelechano, Antonio Ruiz-Corts. An Architectural Discussion on DSPL. 2nd International Workshop on Dynamic Software Product Lines (DSPL08). Limerick, Ireland. 2008.

5. **Carlos Cetina**, Pau Giner, Joan Fons & Vicente Pelechano. Using Variability Models for Developing Self-configuring Pervasive Systems. Workshop on Autonomic and SELF-adaptive Systems (WASELF). Gijon, Spain. 2008.

6. Jose Manuel Marquez Vazquez, **Carlos Cetina**, Francisco Velasco, Luis Gonzalez-Abril, Juan Antonio Ortega. Modelado de caractersticas para itinerarios formativos adaptativos. X Jornadas de ARCA. Sistemas Cualitativos y Diagnosis, Robtica, Sistemas Domticos y Computacin Ubicua (JARCA). Tenerife, Spain. 2008.

7. **Carlos Cetina**, Vicente Pelechano, Sonia Montagud. Inteligencia Ambiental: Protegiendo a los Usuarios Finales de Ellos Mismos. Workshop on Requirements Engineering and Software Environments (IDEAS). Pernambuco, Brasil. 2008.

8. **Carlos Cetina**, Estefanía Serral, Javier Muoz, Vicente Pelechano. Tool Support for Model Driven Development of Pervasive Systems.4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES). Braga, Portugal. 2007

9. **Carlos Cetina**, Javier Muoz, Vicente Pelechano. Software Product Lines Tool Support meets Open Source Software. Proceedings on the Second International Workshop on Open Source Software and Product Lines (OSSPL).

Workshop at Third International Conference on Open Source Systems. Limerick, Irlanda. 2007.

10. Estefanía Serral, **Carlos Cetina**, Javier Muoz, Vicente Pelechano. PervGT: Herramienta CASE para la Generacin Automtica de Sistemas Pervasivos. XII Jornadas de Ingeniera del Software y Bases de Datos (JISBD). Zaragoza Spain. 2007.

8.2.2 Research Visit

The aim of this work was to be influenced and enriched by distinct research streams, works, visions and schools. Thus, along this work there was interaction with the members of the ITEA-MoSiS project (Model-driven development of highly configurable embedded Software-intensive Systems). In particular, the interaction was with Jesús Bermejo from the Telvent Company and with Oystein Haugen from the University of Oslo.

MoSiS is a European project financially supported by the ProFIT program of Ministerio de Industria, Comercio y Turismo in Spain, Tekes in Finland, The Research Council of Norway and The Swedish Governmental Agency for Innovation Systems - VINNOVA.

The main goals in the MoSiS project are:

1. A standardized language for variability modelling and management, supported by tools.

2. Model based approach to variability in extra-functional system properties.

3. Applicability of variability modelling and re-configurable architectures for run-time adapt-ability.

As result of the interaction with Oystein Haugen and the high relation of the MoSiS goals and the work presented in this master thesis, it is planned a research stay of the author of this thesis in Object orientation, Modeling and Language Group (OMS) at the Universty of Oslo. This stay will be hosted by Oystein Haugen from October to December of 2008.

September 15, 2008

8.2.3 Degree Project

Furthermore, the work of this master thesis has been also validated throughout its application in a degree project, where it has been used it in order to resolve a case study. In particular, the degree project in which this work has been applied is the following:

- **Title:** Aplicación de Líneas de Producto a Entornos de Inteligencia Ambiental en la Plataforma Eclipse.
  Student: Sonia Montagud Gregori.
  Reading Date: July, 2007.
  Mark: 10 (MH).

8.2.4 Supporting Tool

The Moskitt Feature Modeler tool (MFM) has been developed in the context of this master thesis. MFM is a feature model editor where the tri-state configuration proposed in this work has been implemented. In a tri-state configuration, features can be set to one of the following states: discarded, deactive or active. Active features conform the initial configuration of the pervasive system, whereas deactive features identified the quiescent components of the system.

This tool has been developed as open source software under the EPL license. MFM was published at http://www.pros.upv.es/mfm on June 23rd, 2008. Since this date, the MFM website has reached 1786 visits and the tool has been downloaded 75 times.

8.3 Assessment

The work presented so far reveals insights combining Software Product Lines and Model Driven Development in the synthesis of Autonomic Pervasive Systems. Although our work exposed some challenges, a close assessment is necessary to reveal some limitations of this work and propose some future work.

8.3.1 Limitations

- Code generation. The SPL presented in this work follows an approach to MDD where code generation is not complete. Although this approach automates cumbersome tasks, it requires some human intervention (e.g., to complete skeletal generated code). This is not exceptional, but is common in other approaches. Nevertheless, future work should address the generation of further code. This would likely embrace the definition of further models.

- Model superimposition. The proposed Realization Model is not a close model. We are currently using and refining the proposed mappings between PervML and FAMA. Furthermore, the PervML method is neither a close research topic. In this context, improved versions of PervML are continuously appearing. Then, new mappings must be defined in case that new conceptual primitives are introduced.

- Validation. The ideas presented in this work were used with a product-line of smart homes that we developed. This worked fine so far. However, further case studies (industrial if possible) are needed to validate our findings.

To overcome these limitations is subject of future work.

8.3.2 Future Research

Most pervasive systems have to interact with highly dynamic environments. Dynamic adaptation has therefore always been an inherent key element of such systems. In order to apply dynamic adaptation for the development of safety-critical and reliable systems, it is necessary that the adaptation behavior is explicitly defined and that the negative as well. Since the specification of the adaptation behavior is a complex and error prone task, a systematic software engineering approach for the development of such systems is required. However, such methodological support for the development of pervasive systems is still in its infancy. To contribute towards this goal, our work presents an agenda for future research.

- Current end-user programming techniques are mainly focused on providing appropriate metaphors to user skills [101, 102, 103], whereas none of these techniques addresses the evolution of user needs or devices. We are working in a design method for adaptive smart homes where end-users and technical designers participate cooperatively. End-users contribute with their context and domain knowledge, while designers introduce their technical background to preserve the quality of the system. We complement this method with a specification technique so that both end-users and designers configure the systems in terms of features.

- For autonomic pervasive systems, it is indispensable to have a means to analyze the adaptation behavior already at design time and to guarantee certain properties. Therewith this model-driven approach makes it possible to identify reasonable configurations in an early stage of the development process without first implementing them. As for any other software engineering approach it is particularly possible to analyze and to predict the quality of the adaptation behavior to enable systematic control of the development process. We are working on extend the feature modeling technique with a set of metrics to evaluate the adaptation capabilities of the smart home. In particular, we are working on structural metrics, which can be calculated once the designers have defined the family scope, and run-time metrics which can be applied each time the system reconfigures itself.

- In this work, we have defined how a set of components that make up an architectural or design pattern dynamically cooperate to change the software configuration to a new configuration. To achieve this goal, We have applied the Quiescent software adaptation pattern which models how the software components and interconnections can be changed. Further research includes effective approaches for automatically evolving software architectures, in particular how to maintain the service sate while adaptation is taking place, and Quality of Service issues during software adaptation.

- Aspects related to other contextual triggers are not addressed in the proposed

approach. A forthcoming work will focus on complement this approach with context aware capabilities. Currently, the PervML framework is been extended with a context layer where location and user-personalization are taken into account.

- Tool Support. Some tools were implemented to support our ideas (e.g., MFM, PervML Framework, etc). Although we spent a great amount of effort on them to work, still they require further work to be used by a regular customer.

Bibliography

[1] Paul Clements and Linda Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2002.

[2] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.

[3] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):94–104, 1999.

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

[5] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, 2003.

[6] Mike Mannion. Using first-order logic for product line model validation. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 176–187, London, UK, 2002. Springer-Verlag.

[7] D. Benavides, Ruiz A. Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[8] T. Lemlouma and N. Layaida. Context-aware adaptation for mobile devices. *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, pages 106–111, 2004.

[9] Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. Automating product-line variant selection for mobile devices. *Software Product*

Line Conference, 2007. SPLC 2007. 11th International, pages 129–140, 10-14 Sept. 2007.

[10] M. Douglas McIlroy. Mass-produced software components. In J. M. Buxton, Peter Naur, and Brian Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, October 1968.

[11] D.L. Parnas. On the design and development of program families. *Software Engineering, IEEE Transactions on*, SE-2(1):1–9, March 1976.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[13] F. van der Linden. *Lecture Notes in Computer Science*, volume 2290. Springer, Bilbao, Spain, October 3-5, 2001 2002.

[14] P. Donohoe. Number ISBN 0-7923-7940-3. Denver, Colorado, USA, August 28-31.

[15] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[16] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:2004, 2003.

[17] L. Northrop. SEI's Software Product Line Tenets. *IEEE Software*, 19(4):32–40, July/August 2002.

[18] G. Chastek and J.D. McGregor. Guidelines for developing a product line production plan. Technical report, CMU/SEI, June 2002.

[19] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[20] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[21] Stephen J. Mellor and Anthony N. Clark and Takao Futagami. Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, 20(5):14–18, 2003.

[22] Amílcar Sernadas, Cristina Sernadas, and Hans-Dieter Ehrich. Object-oriented specification of databases: An algebraic approach. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 107–116, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[23] Ralf Jungclaus, Gunter Saake, Thorsten Hartmann, and Cristina Sernadas. TROLL: a language for object-oriented specification of information systems. *ACM Trans. Inf. Syst.*, 14(2):175–211, 1996.

[24] Michael Rohs and Jürgen Bohn. Entry points into a smart campus environment " overview of the ethoc system. In *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 260, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Object Management Group. Unified Modeling Language: Superstructure version 2.1.1. OMG Specification, February 2007.

[26] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[27] Stuart Kent. Model driven engineering. In *Proceedings of the Third International Conference Integrated Formal Methods (IFM'2002)*, 2002.

[28] Ivan Kurtev. *Adaptability of Model Transformations.* phdthesis, IPA, 2005. ISBN 90-365-2184-X.

[29] Jean Bzivin, Nicolas Farcet, Jean marc Jzquel, Benot Langlois, and Damien Pollet. Reflective model driven engineering. pages 175–189. Springer, 2003.

[30] J. Bzivin. In search of a basic principle for model driven engineering. *UPGRADE*, 1:15–24, 2004.

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

[32] Joaquin (Hrsg.) Miller and Jishnu (Hrsg.) Mukerji. Mda guide version 1.0.1, 2003. Letzte Änderung am 12. Jun. 2003, besucht am 15. Mai 2008.

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

[34] J. Jann, L. M. Browning, and R. S. Burugula. Dynamic reconfiguration: Basic building blocks for autonomic computing on ibm pseries servers. *IBM Syst. J.*, 42(1):29–37, 2003.

[35] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Syst. J.*, 42(1):60–76, 2003.

[36] Stephen A. Jarvis, Daniel P. S pooner, Helene N. Lim Choi Keung, Justin R.D. Dyson, Lei Zhao, and Graham R. Nudd. Performance-based middleware services for grid computing. *ams*, 0:151, 2003.

[37] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, and Sandeep K.S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 01(3):33–40, 2002.

[38] Gordon S. Blair, Geoff Coulson, Lynne Blair, Hector Duran-Limon, Paul Grace, Rui Moreira, and Nikos Parlavantzas. Reflection, self-awareness and self-healing in openorb. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 9–14, New York, NY, USA, 2002. ACM.

[39] V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM Syst. J.*, 42(1):98–106, 2003.

[40] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing web server performance with autotune agents. *IBM Syst. J.*, 42(1):136–149, 2003.

[41] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, Sept. 1991.

[42] David Wright, Elena Vildjiounaite, Ioannis Maghiros, Michael Friedewald, Michiel Verlinden, Petteri Alahuhta, Sabine Delaitre, Serge Gutwirth, Wim Schreurs, and Yves Punie. Safeguards in a world of ambient intelligence (swami) deliverable d1. the brave new world of ambient intelligence: A state-of-the-art review, June 2005. A report of the SWAMI consortium to the European Commission under contract 006507.

[43] Friedemann Mattern. *Ubiquitous Computing: Scenarios from an informatised world*, pages 145–163. Springer-Verlag, 2005.

[44] Jochen Burkhardt, Thomas Schaeck, Horst Henn, Stefan Hepper, and Klaus Rindtorff. *Pervasive Computing: Technology and Architecture of Mobile Internet Applications*. Addison-Wesley, April 2002.

[45] Uwe Hansmann, Lothar Merk, Martin S. Nicklous, and Thomas Stober. *Pervasive Computing Handbook*. Springer-Verlag, 2001.

[46] O.A. Rawashdeh and J.E. Lumpp. A technique for specifying dynamically reconfigurable embedded systems. *Aerospace Conference, 2005 IEEE*, pages 1–11, March 2005.

[47] ROSES. Robust self-configuring embedded systems. http://www.ece.cmu.edu/ koopman/roses/.

[48] Depaude-Project Webpage. Dependability for embedded automation systems in dynamic environment with intra-site and inter-site distribution aspects. http://www.esat.kuleuven.be/electa/depaude/.

[49] Jamieson M. Cobleigh, Leon J. Osterweil, Alexander Wise, and Barbara Staudt Lerner. Containment units: a hierarchically composable archi-

tecture for adaptive systems. *SIGSOFT Softw. Eng. Notes*, 27(6):159–165, 2002.

[50] Wills, L.M., Kannan, S., Sander, S., Guler, M., Heck, B., Prasad, J.V.R., Schrage, D., Vachtsevanos, G. A prototype open control platform for reconfigurable control systems. *Software-Enabled Control: Information Technologies for Dynamical Systems*, pages 63–84, May 2003.

[51] Ji Zhang and Betty Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM.

[52] Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA, 2005. ACM.

[53] Sandeep S. Kulkarni and Karun N. Biyani. Correctness of component-based adaptation. Technical Report MSU-CSE-04-2, Department of Computer Science, Michigan State University, East Lansing, Michigan, January 2004.

[54] Elisabeth A. Strunk. *Reconfiguration Assurance in Embedded System Software.* PhD thesis, University of Virginia.

[55] M. Trapp. *Modeling the Adaptation Behavior of Adaptive Embedded Systems.* PhD thesis, Technical University of Kaiserslautern, 2005.

[56] Mario Trapp, Rasmus Adler, Marc Förster, and Janosch Junger. Runtime adaptation in safety-critical automotive systems. In *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*, pages 308–315, Anaheim, CA, USA, 2007. ACTA Press.

[57] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93, Jan 2000.

[58] Gme. http://www.isis.vanderbilt.edu/projects/gme/.

[59] Andreas Beicht. Entwicklung eines frameworks zur entwicklung und analyse adaptiver eingebetteter systeme. Master's thesis, TU Kaiserslautern, 2007.

[60] Klaus Schneider, Tobias Schuele, and Mario Trapp. Verifying the adaptation behavior of embedded systems. In *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 16–22, New York, NY, USA, 2006. ACM.

[61] Rasmus Adler, Marc Forster, and Mario Trapp. Determining configuration probabilities of safety-critical adaptive systems. *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, 2:548–555, May 2007.

[62] Software product-family engineering. *5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, 3014, 2004.

[63] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[64] Jaejoon Lee and Kyo C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. *splc*, 0:131–140, 2006.

[65] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. *Software Product Line Conference, 2006 10th International*, pages 21–24, Aug. 2006.

[66] P. Trinidad, , A. Ruiz-Cortés, and J. Pe na. Mapping feature models onto component models to build dynamic software product lines. *International Workshop on Dynamic Software Product Line*, 2007.

[67] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu. Towards requirements-driven autonomic systems design. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM.

[68] Christian Tischer, Andreas Muller, Markus Ketterer, and Lars Geyer. Why does it take that long? establishing product lines in the automotive domain. *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 269–274, 10-14 Sept. 2007.

[69] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 233–242, 10-14 Sept. 2007.

[70] Carlos Cetina, Joan Fons, and Vicente Pelechano. Applying Software Product Lines to Build Autonomic Pervasive Systems. *Software Product Line Conference, 2008. SPLC 2008. 12th International*, 8-12 Sept. 2008.

[71] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *Software, IEEE*, 15(6):37–45, Nov/Dec 1998.

[72] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. Model-driven software product lines. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 126–127, New York, NY, USA, 2005. ACM.

[73] Douglas C. Schmidt, Andrey Nechypurenko, and Egon Wuchner. MODELS'05 Workshop "MDD for Software Product-lines: Fact or Fiction?". http://www.geocities.com/andreynech/ MDDandProductLinesWorkshop.html, 2005.

[74] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature oriented model driven development: A case study for portlets. *icse*, 00:44–53, 2007.

[75] Javier Muñoz and Vicente Pelechano. Applying software factories to pervasive systems: A platform specific framework. In *ICEIS (3)*, pages 337–342, 2006.

[76] Javier Muñoz, Vicente Pelechano, and Carlos Cetina. Implementing a pervasive meeting room: A model driven approach. In *IWUC*, pages 13–20, 2006.

[77] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Comput. Networks*, 51(2):456–479, 2007.

[78] Yijun Yu, John Mylopoulos, Alexei Lapouchnian, Sotirios Liaskos, and Julio Cesar Sampaio do Prado Leite. From stakeholder goals to high-variability software designs. Technical report, University of Toronto, 2005. Available at: ftp://ftp.cs.toronto.edu/csrg-technical-reports/509/.

[79] Marcos Didonet Del Fabro, Jean Bzivin, and Patrick Valduriez. Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium*, 2006.

[80] Felix Loesch and Erhard Ploedereder. Optimization of variability in software product lines. *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 151–162, 10-14 Sept. 2007.

[81] Eclipse Foundation. ATL Model Transformation Language website. http://www.eclipse.org/m2m/atl/.

[82] D. Marples and P. Kriens. The open services gateway initiative: an introductory overview. *Communications Magazine, IEEE*, (12):110–114, Dec 2001.

[83] Carlos Cetina, Estefana Serral, Javier Munoz, and Vicente Pelechano. Tool support for model driven development of pervasive systems. *mompes*, 0:33–44, 2007.

[84] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez. Fama framework. In *Software Product Line Conference, 2008. SPLC 2008. 12th International*.

[85] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.

[86] Choonhwa Lee, D. Nordstedt, and S. Helal. Enabling smart spaces with osgi. *Pervasive Computing, IEEE*, 2(3):89–94, July-Sept. 2003.

[87] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[88] Hassan Gomaa. Designing software product lines with uml 2.0: From use cases to pattern-based software architectures. *splc*, 0:218, 2006.

[89] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[90] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[91] Hassan Gomaa. Architecture-centric evolution in software product lines. In *In ECOOP-ACE 05*, 2005.

[92] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2005.

[93] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with Uml*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[94] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.*, 5(2):128–138, 1979.

[95] Hassan Gomaa and Mohamed Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. volume 0, page 79, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[96] Hassan Gomaa. A software modeling odyssey: Designing evolutionary architecture-centric real-time systems and product lines. In *MoDELS*, pages 1–15, 2006.

[97] Nelly Bencomo, Gordon Blair, and Robert France. Model-driven software adaptation report on the workshop m-adapt at ecoop 2007. *Object-Oriented Technology. ECOOP 2007 Workshop Reader*, pages 132–141, 2008. Springer, LNCS.

[98] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *Software Engineering, IEEE Transactions on Software Engineering*, pages 1293–1306, 1990.

[99] J. Kramer and J. Magee. Analysing dynamic change in software architectures: a case study. *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on Configurable Distributed Architecture*, pages 91–100, 1998.

[100] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems*, 2007.

[101] Jan Humble, Andy Crabtree, Terry Hemmings, Karl-Petter kesson, Boriana Koleva, Tom Rodden, and Pr Hansson. Playing with the bits" user-configuration of ubiquitous domestic environments. *Ubicomp 2003*, pages 256–263, 2003.

[102] Hague, R., et al. Towards pervasive end-user programming. *UbiComp 2003*, pages 169–170, 2003.

[103] Khai N. Truong, Elaine M. Huang, and Gregory D. Abowd. Camp: A magnetic poetry interface for end-user programming of capture applications for the home. *UbiComp 2004*, pages 143–160, 2004.

[104] T. Dean Hendrix, II James H. Cross, Saeed Maghsoodloo, and Matthew L. McKinney. Do visualizations improve program comprehensibility? experiments with control structure diagrams for java. *SIGCSE Bull.*, 32(1):382–386, 2000.

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

[106] J. Hess W. Novak K. Kang, S. Cohen and S. Peterson. Featureoriented domain analysis (foda) feasibility study. technical report cmu/sei-90-tr-21. Technical report, Software Engineering Institute, Carnegie Mellon University, November 1990.

[107] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the rseb. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 76, Washington, DC, USA, 1998. IEEE Computer Society.

[108] Detlef Streitferdt Matthias Riebisch and Ilian Pashov. Modeling variability for object-oriented product lines. In *ECOOP 2003Workshop Reader, volume 3013 of Lecture Notes in Computer Sciences. SpringerVerlag*, 2004.

[109] Simon Helsen Krzysztof Czarnecki and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. In *Software Process: Improvement and Practice, 10(1):729*, 2005.

[110] Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM.

[111] Robert Spence. *Information visualization.* Addison-Wesley., Harlow, England, 2000.

[112] B. A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. *SIGCHI Bull.*, 17(4):59–66, 1986.

[113] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. Principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, June 1993.

[114] Pure Systems. Pure::Variant website. http://www.pure-systems.com/.

[115] Big Lever. Gears. http://www.biglever.com/solution/solution.html.

[116] Michalis Demetriou and Roberto E. Lopez-Herrejon. Feature designer - a feature modeling tool for .net. In *1st International Workshop onVisualisation in Software Product Line Engineering*, 2007.

Appendix A

MOSKitt Feature Modeler

A good notation can make models more comprehensible [104]. In order to handle models effectively, tool support should be provided. Tools should support an adequate notation providing facilities for the representation, creation and edition of models. In addition, editors should provide mechanisms to manage the complexity when models start to grow. This becomes a must when modeling complex systems.

In the present work, we are particularly interested in Feature Models. Different notations and tools exist for this kind of models. However, tool support for the edition of Feature Models has some limitations referring to complexity handling. When models become populated with many features, the readability of models is affected. This hinders the use of such tools for modeling large-scale systems. In addition, these tools usually lack customization support, so the user preferences are not considered.

We propose some visualization techniques to make editors scale as the model grows, considering also user customization. We have analyzed the particular visualization needs demanded by Feature Modeling to detect some points of improvement. Finally we have developed Moskitt Feature Modeler (MFM), a graphical editor that addresses the detected limitations offering editing capabilities suitable for the feature modeling of large systems. In addition, the developed tool has been integrated in a tool set for the support of software development process.

The remainder of the appendix is structured as follows. In Section A.1 particular requirements to support complexity handling in feature Models are detected.

Section A.2 introduces MFM where we have applied visualization techniques to handle complex models. Section A.3 presents the interoperability capabilities of MFM. Related work is presented in Section A.4. Finally, Section A.5 presents conclusions and further work.

A.1 Requirements for Feature Visualization

CASE tools in Software Engineering help to program in terms of the design intent rather than the underlying computing platform. However, as stated in [105], many CASE tools suffer from the inability to scale to handle complex, production-scale systems in part because their "one-size-fits-all" graphical representations are too generic and non-customizable.

In the Feature Modeling area, different notations and tools have emerged for the definition of such models. These tools are focused on giving basic edition support to a certain notation but they are not suited to large-scale systems. From analyzing the current state of the art in feature model editors, we have detected some requirements that can improve the complexity handling when feature models become highly populated. These requirements are detailed below.

A.1.1 Using layout to reflect the structure

When diagramming, the position of the different elements can provide an insight of the underlying information structure. Elements presented physically close are also considered "close" in a semantic sense when observed.

It is important to allow users to organize their diagrams according to their own criteria. However, most of the Feature Models define a hierarchical structure that can be represented by a tree. It is interesting to allow not only a vision of each branch in depth –features and their sub-features– but also consider horizontal comparison –e.g., to check which elements are at the same depth level– and allow an easy detection of special nodes –e.g.,root or leaf nodes– to better appreciate the topology.

Current tools represent Feature Models using unfoldable nested nodes –normally an icon accompanied by a text label– such as the folder view most file system

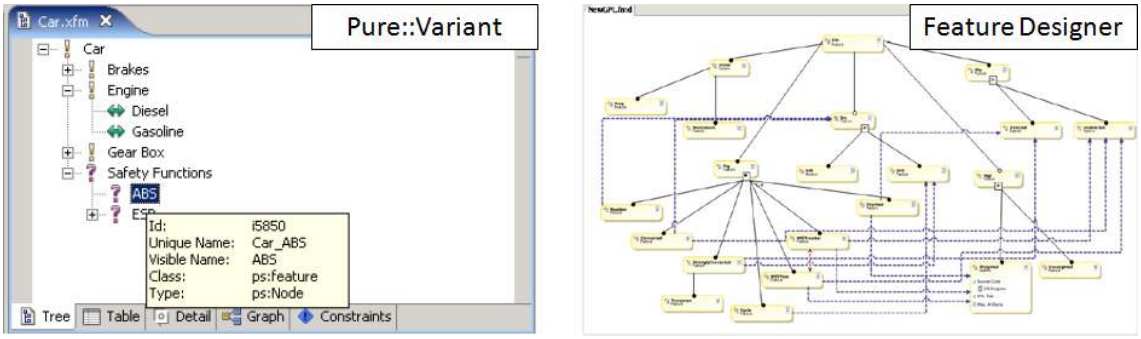

Figure A.1: Foldable node and diagram based editors.

explorers offer –as depicted at the left hand side of Figure A.1–. Although it is well suited for many uses, a more diagrammatic view is required –illustrated at the right hand side of Figure A.1–. There is a need for a view that offers an idea of the overall information structure at a first sight. In order to do so, a two-dimensional tree representation is proposed. This representation should be balanced and respect a clear visualization for the elements that are situated at the same depth.

Facilities for the identification of special nodes should be also provided. This should be done with modifications that do not invalidate the notation constraints –e.g., changing background color for figures but respecting their shape–.

A.1.2 Nesting capabilities

One of the most common techniques in programming to handle complexity is packaging. Some modeling techniques adopted similar solutions to handle complexity at modeling level. For example, graphical notations such as UML Class Diagram support a package concept. In this way, a system where sub-systems are detected can be described in a modular way. This improves the understandability of the system since it can be seen at different levels of detail –from an overview of the system to a detailed view of each part–.

In Feature Models, some features can be considered sub-features –as we stated before, these kind of models usually are structured as a tree–, thus mechanisms should be offered to allow nesting capabilities for features.

Hiding the children of some features can be useful to reduce detail when diagramming. However, some feedback should be provided to (1) indicate that a feature is

collapsed, and (2) give an intuitive idea of the number of hidden elements.

Feature Editors, since they are commonly based on foldable nodes, provide a homogeneous support for nesting –all nodes can be nested– but they do not offer feedback about the number of hidden nodes. However, for the construction of a diagrammatic editor to define feature models, this requirement should be faced.

A.1.3 Support for multiple notations

In the Feature Modeling community, several graphical notations have been proposed. Although these notations can be considered equivalent to some extent, depending on the desired detail level, some notations can be preferred to others.

Different notations provide different representations for features and their relationships. Each notation provides a different amount of information. Some notations represent cardinalities in relationships by means of graphical elements, some notations use text labels indicating cardinality boundaries using numbers, and other notations do not represent cardinalities at all.

By choosing an adequate notation, the amount of information displayed in a diagram can be managed. A notation with much detail can provide very useful information but it can also lead to cluttered diagrams that difficult the overall model understandability.

In order to manage how much information is exposed in diagrams in order to handle complexity, a tool for the definition of feature models should support different notations. In this way, the tool can offer the detail a specific user or project needs.

A.1.4 Customization

The role the user plays in the above requirements should be empowered. The user should be in control of the applied layout, how features are nested and which notation is in use.

Mechanisms for changing these aspects –and undoing changes– should be provided. The user should be able not only to change these parameters, but also to select the affected parts of the diagram. Facilities should be provided for both, gen-

eral and particular customization. On the one hand, changes should be applicable simultaneously to all element easily. On the other hand, fine-grained customization should be allowed.

Current tools offer a limited support for customization. Generally, the layout is predefined, all nodes are nestable and only one particular notation is supported. This prevents their use for large projects where different participants with particular preferences are involved.

A.1.5 User guidance

Some of the previous requirements –such as changes in the layout or the notation– involve a transition from an existent representation of the model to a new one. In order to avoid users from "being lost" in this transition, guidance mechanisms should be provided.

When a visualization changes, finding an element in the new representation, can suppose an effort for the user. This effort can be alleviated by (1) offering feedback to the user about the performed operations and (2) performing the operations in a gradual way.

Tool support should provide guidance support for the user. On the one hand, feedback about user actions should be provided in an unobtrusive way avoiding distraction. On the other hand, by making changes gradual, the user can be guided seamlessly from one representation to the new one.

In graphical terms, feedback can be provide by color changes, and animation techniques can be used for making operations –such as applying a layout– gradual.

A.2 Moskitt Feature Modeler

Models are valuable documentation assets since they capture relevant information of a system. However, they can play a more relevant role in the development process of a system. MDE proposes the use of models as central assets for system development.

Provided that models are defined using precise semantics, they can become machine-processable. This enables the automatic manipulation of models to ob-

tain new assets –other models, software systems, documents, etc.–. In order to provide an effective MDE development process, good tool support is required to handle models.

Feature Modeling has an important role in the software industry. Although an isolated modeling tool for feature models can be interesting by its own, the integration with a MDE method and tools for software development provides new value. We have integrated the presented feature model editor in Moskitt.

Moskitt[1] is an open source modular modeling set of tools to support the development process defined by the Infrastructures and Transport Ministry of the Generalitat Valenciana in Spain. The tool is based on Eclipse and provides support for model management –graphical edition, persistence, cooperative work, etc.–. It also allows to establish relationships between models –generation of models and production of traces, dependencies, consistency checks, etc.– and the production of new assets taking the models as input.

Moskitt supports different kind of models. UML models are supported. These diagrams are based on the full UML 2.0 metamodel defined by the OMG. In addition to UML, other models are supported to cover particular needs. Moskitt provides support for the modeling of relational database schemata –that can be derived from UML models–. It also provides a Requirements Editor for the generation and maintenance of a requirements catalog and support for traceability between requirements and the rest of models. User Interface and Business Process modeling is also supported by the tool.

Moskit Feature Modeler (MFM) is the open source feature model editor of Moskitt. MFM shares several technologies with Moskitt editors and takes advantage of some infrastructure components defined by Moskitt –such as a manager for model-to-model transformations–. Thanks to the support for defining relationships among models, Feature models can be used in conjunction with the existing editors –without modification– to enable their use for Product Line Engineering.

As a result, MFM has been integrated in the Moskitt tool set, providing edition

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

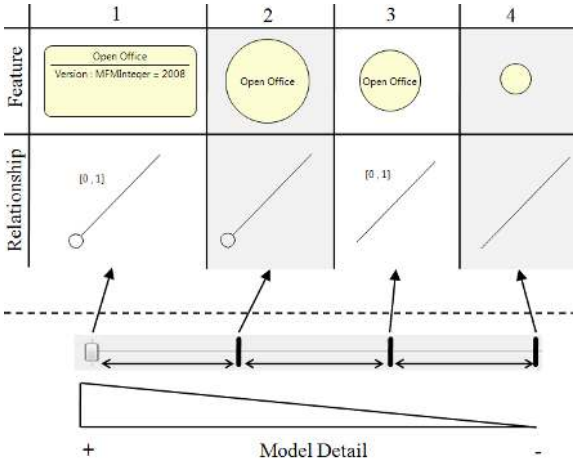

Figure A.2: Supported notations in MFM.

capabilities for Feature Models that can be used along the software developent process. Therefore, a new opportunity to the MDE community for the development of new tools and extensions to operate on those models is provided.

Due to Moskitt and MFM are applied in real industrial scenarios, visualization capabilities takes special importance to handle large feature models. The following Sections introduces the visualization techniques we have incorporated to MFM to handle model complexity.

A.2.1 Customizing the Notation Style

In 1990, Kang et al. [106] proposed feature models for the first time. However, despite years of research, there is no consensus on the modeling artifacts allowed in feature models and many extensions have been proposed since then. First, the original feature model notation called FODA [106] was proposed. Later, Feature-RSEB [107] was presented as a FODA extension with an additional relationship. Finally, Riebisch et al. [108] and Czarnecki et al. [109]. proposed cardinalitybased feature models where cardinalities were introduced.

Despite the lack of consensus on feature models, Pierre-Yves et al. have proposed a generic formalization of the syntaxis and semantics of feature model [77]. According with the results of their work, we have incorporate to MFM **support to multiple graphical notations**. The MFM tool supports both cardinality and

relationship-decorations notations and also, MFM introduces a simplified notation for visualization purposes. Users can dynamically change the graphic notation of feature models.

MFM supports customizing the notation at any time between the following feature representations (see top of Figure A.2).

1. **Feature with Attributes.** Features are graphically represented by means of rectangles. These rectangles are composed of two compartments. The top compartment holds the feature name and the bottom compartment holds the features attributes. These features attributes follows the pattern: <name>:<type>=<value>

2. **Rounded Feature.** Features are graphically represented by means of ellipses. The feature name is at the ellipse center, whereas feature attributes are not shown.

3. **Fixed Feature.** Features are represented as Rounded features which diameter depends of the feature name length.

4. **Simplified Feature.** Features are graphically represented by means of ellipses. Neither the feature name is visible, nor the feature attributes. The ellipse diameter is set to a constant.

MFM also supports customizing relationship notation as follows:

1. **Cardinality-Graphic Relationship.** Relationships are represented by means of decorated lines and a flotating label. The line decoration indicates the type of relationship. Optional relationships are decorated with a white ellipse and mandatory relationships are decorated with a black ellipse. The label follows the pattern [min, max] to indicate the minimum and maximin cardinality of the relationship. Both label and decoration are synchronized between them.

2. **Graphic Relationship.** Relationships are represented by means of decorated lines.

3. **Cardinality Relationship.** Relationships are represented by means of lines and a flotating label.

4. **Simplified Relationship.** Relationships are represented only by means of lines.

Customizing in the Large vs Customizing in the Small

MFM enables users to dynamically switch the graphic representation of features and relationships. At any time, users can select the features and relationships notation style and MFM changes the elements representation. Generally, users set the notation style according to their preferences and then they apply the style to all the model elements. This is what we call **customizing in the large**. However, users may prefer different representations in particular cases. For example, when feature notation is set to Rounded *Feature the attributes* are not visible and the user may change the notation of a concrete feature to Feature with Attributes with the aim of editing feature attributes. To support this user behavior, we complement Customizing in the Large wit Customizing in the Small. **Customizing in the small** enables users to change the notation style of individual elements. The combination of both approach enables users to globally set the the notation style and them configure individual elements at any time.

Configuring the detail level of models

Feature models provides domain information of model elements and they also provide holistic information from the elements structure. Some of the notation styles provide domain information (such as Feature with Attributes and Cardinality-Graphic Relationships), whereas other notations focus on structure information (such as Simplified Features and Simplified Relationships).

To help users selecting the notation style best suited to each type of information, we have incorporated to MFM a model detail tab which provides an horizontal slide (see bottom of Figure A.2) This slide defines four predefined states with combinations of notations styles. The most-left state highlights domain information, while the most-right state highlights structure information. The states combine the notations styles as follows:

1. Feature with Attributes and Cardinality-Graphic Relationship. Feature names,

attributes and cardinalities have a graphic representation. This combination generates detailed views where the domain information can be edited graphically. However model complexity can hide model structure.

2. Rounded Feature and Graphic Relationship. This combination hides features attributes and relationships cardinalities.

3. Fixed Feature and Cardinality Relationship. This representation set the feature shapes dimension according to the feature name length. When features with short names are resized, they may be confused with relationships decorations. To avoid this confusion, relationships decorations are replaced by cardinality based representations.

4. Simplified Feature and Simplified Relationship. This combination hides all the domain information, highlighting the structure information.

The model detail slide provides users with a *quickly and intuitive mechanism to change the whole model representation*. Moving the slide form left to right the user decrease the model detail, highlighting the model structure. On the other hand, moving the slide form right to left, the user increases the model detail.

A.2.2 Visualizing Model Structure

Previous Section presented the MFM capabilities to dynamically change the notation style. MFM incorporates predefined combinations of notation styles which helps users to focus on model structure. However, when model complexity increases to industrial size problems, dealing with model structure requires advanced visualization techniques. This section presents two visualization techniques incorporated on MFM to highlight model structure on complex feature models.

Users structure elements on models using their own criteria (top of Figure A.3), but this approach is limited as it neither guarantees any (1) semantic distribution, nor follows any (2) standard layout. Semantic distribution (1) uses model elements distribution in order to provide semantic information instead of only improve model legibility. For example, features can be distributed on rows with the semantic that

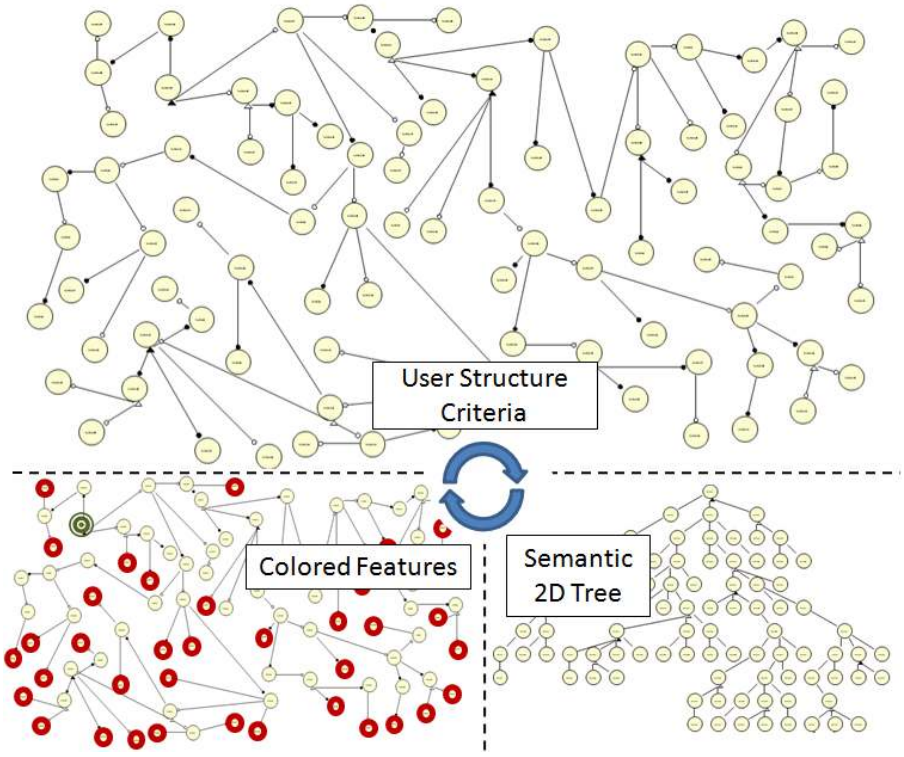

Figure A.3: Colored Features and Semantic 2D Tree Techniques

a in the nth row all its features have (n-1) parent features. A standard layout (2) promotes several users working with the same model. As long as all users share a standard layout, they will have facilities to find model elements. For example, always drawing the root feature on top of all the other features.

However, from the point of view of users, their own structure criteria can be as important as other semantic or a standard criteria. Therefore, We have introduced in Moskit Feature Modeler visual techniques to provide users with semantic and standards distributions. Furthermore, these techniques are applied without loosing the user structure criteria.

The structure visualization technique creates dynamic views over models to identify key elements and the inherent model structure. We consider a feature model as a 2D tree where features are nodes, and relationships (optional, mandatory, or and alternative) are directed connection between nodes. The root feature of this tree does not have any incoming connection with other features. The leaf features do not have any outcoming connection to other features. Finally, each level of the tree includes features that have the same number of parent features.

To highlight the key elements and structure of the features tree, we have incorporated to Moskitt Feature Modeler the following visualization techniques:

- **Colored Features.** The border color of a feature depends on the role that the feature plays in the tree (see bottom left of Figure A.3). The root feature has a red border, while leaf features have a green border. This technique helps users to identify the root and leafs of a tree meanwhile the user structure criteria is not affected. However, this technique does not structure features according to tree levels.

- **Semantic 2D Tree.** Model elements are graphically redistributed to conform a tree structure (see bottom right of Figure A.3). A layout algorithm takes as input a feature model and generates a graphic tree according to the role that each feature plays in the model. The root feature is always positioned at the top, while leaf features are positioned at the bottom. Finally, the rest of features are positioned on aligned rows according to their number of parent features.

  When the layout algorithm is applied, the position of features and relationships may change at whole. For example, applying this algorithm to the top model of Figure A.3 generates the bottom right model of the same Figure. Comparing source the model with the generated model, almost none of the model elements preserve its position. If this algorithm is applied in an atomic step, the user may loose the connection between source model elements and generated model elements. To empower user guidance, the layout algorithm provides a graphical animation which shows how the model elements are graphically reallocated. This animation helps users to trace the new position of model elements.

The application of these visualization techniques do not exclude the user criteria to organize feature models. Colored features can be applied simultaneously with the user criteria. In the case of Semantic 2D Tree, MFM supports applying this technique to observe the resulting structure and then undo the changes and go back to the user model organization.

A.2.3 Feature Explosion with Visual Guidance

Previous Sections describe the capabilities of MFM to work with model structure. SectionA.2.1 argues the use of detail levels to focus on domain information or structure information. Then, SectionA.2.2 present visualization techniques to highlight model structure on complex feature models. In this Section, we introduce Feature Explosion, a visual technique to structure feature models in submodel, providing users with another mechanism to handle model complexity.

Figure A.4 compares Feature Explosion with the classic feature modeling approach. Top of Figure introduces the OpenOffice.org (OOo) example that we use to compare both approaches. OOo is an open-source office software suite for word processing, spreadsheets, presentations, graphics, databases and more. According to the OOo installation wizard, we notice that OOo is structured on Program Modules (such as Word, Calc, Draw...) and Optional Components common to all Program Modules (such as spell checker languages). Furthermore, each one of the program modules features its specific optional components. For example, the Calc Module have specific components such as Euro Tool, Solver or Formula Analyzer.

Following the classic feature modeling approach, the whole OOo example is represented using a single feature model. Program Modules and Optional Components are modeled as children of the OpenOffice.org root feature (see the middle of Figure A.4). Specific components are also represented in the same model as children of their Program Module For example, the specific components of the Draw and Calc Program Modules are remarked with a grey area on Figure A.4. However, gartering all this information in a stand alone model affects model compressibility and manipulation. The more complex models are, the more important become visualization techniques.

To handle model complexity, we introduce the Feature Explosion technique complemented with visual feedback. The goal of Feature Explosion is structure models by means of submodels. Every Feature can be exploited on a submodel with its children features. For instance, in the OOo example we identify Program modules (such as Word, Calc or Draw) as good candidates to be exploited on submodels.

As a consequence of using Feature Explosion technique, several submodels can

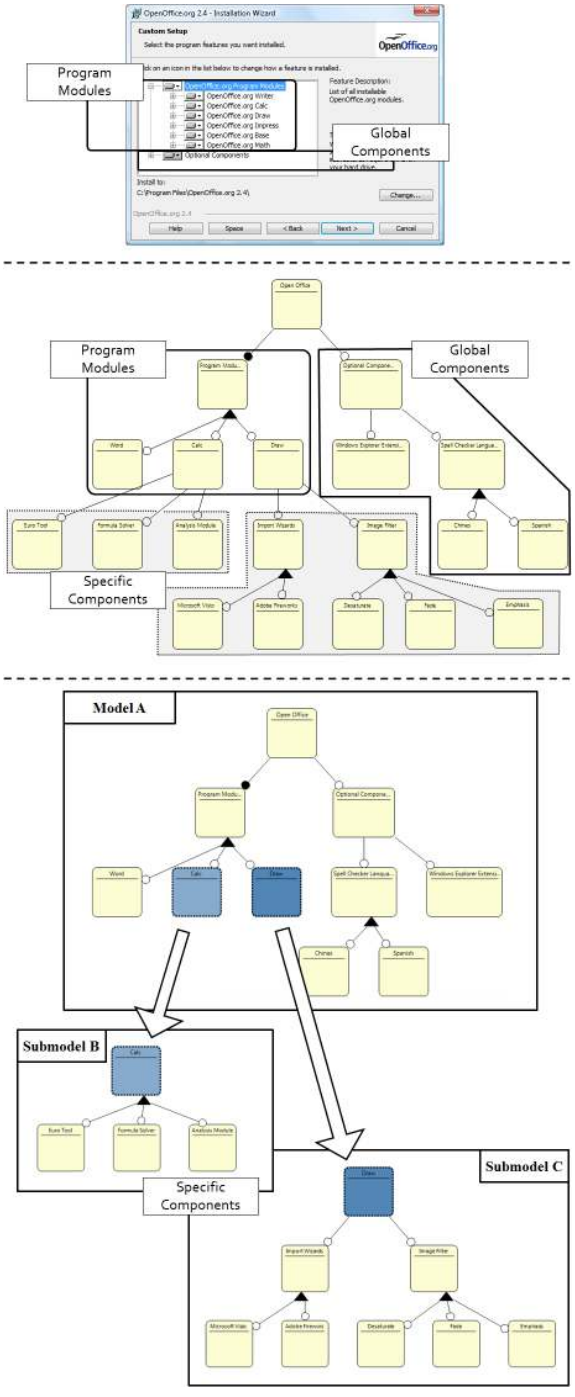

Figure A.4: Feature Explosion Technique

be open at the same time. To enhance model-submodel connection, we apply visualization techniques at both models and submodels:

- **Submodel Connection.** Each submodel is automatically created with a clone of the exploited featured which generated the submodel. This clone acts as the root of the submodel and all other submodel features are proxy's children. The clone is created with visualization purposes only, it is a logic element which has not persistence. For example, Feature *Draw* is exploited on Submodel C, which has a *Draw* clone playing the role of submodel root. All *Draw* subfeatures are modeled as clone children in the Submodel C.

- **Model Connection.** This technique complements exploited features with summarized information about its children. This information is related with feature submodel population and it is expressed by means of changes in feature visual representation.

  - Feature with submodel. The graphic representation differentiates exploited features of non-exploited features. Non-exploited features are represented using a solid border shape, whereas exploited features are represented using a discontinued border shape. The aim of this differentiation is help users to notice which features are associated to submodels and which are not. For example, at bottom of the Figure A.4 the features Calc and Draw of model A are represented using a discontinued border shape.

  - Submodel cardinality. Exploited features differentiate of non exploited by means of the graphic representantion. Furthermore, the graphic representation reflects the submodel cardinality of exploited features. The shape fill color changes its intensity according to the submodel population. Initially, exploited features are render using a light blue color. The more more elements have the feature submodel, the more dark becomes the fill color. For example, at bottom of the Figure A.4 the features of model A are rendered with different color intensities. The Calc feature (which have three submodel elements) is filled with a lighter color than

Figure A.5: MFM-FMP Model Comparison

Draw feature (which have seven submodel elements).

Feature Explosion can be simultaneously applied with the visualization techniques presented in previous sections. All these techniques and their possible combinations enable users to have a better understanding for feature models, specially of complex models.

A.3 MFM-FMP Interoperability

The Feature Modeling Plug-in [110] (FMP) is an Eclipse plug-in for editing and configuring feature models. FMP can be used standalone, in Eclipse, or in Rational Software Modeler or Rational Software Architect. Since September 2006, FMP is open source under Eclipse license, however the project has been completed and the tool is no longer maintained.

MFM can export feature models that are compatible with FMP. The Moskitt trnasformation engine incorporates a Model to Model (M2M) transformation which takes as input MFM models and generates FMP models. The resulting FMP models are compliant with FMP metamodel and they can be edited using FMP plugin. The M2M rules have been implemented using the INRIA ATL [81] transformation language and tools. These rules are also available as an ATL standalone file and the

rules file can be downloaded from the MFM web site.

Figure A.5 shows the non-graphical representation of a feature model using MFM (left side of Figure). Taking this model as input, the M2M transformation generates a FMP compliant model. The generated model can be edited using the FMP plugin (right side of Figure).

This M2M bridge enables users to maintain their current implementations build on top of FMP metamodel, meanwhile they take advantage of the visualization techniques incorporated in MFM to manipulate feature models.

A.4 Related work

Graphical modeling tools represent and manipulate models. If we consider models as data, *Information Visualization* techniques [111] can be applied to improve the comprehension of these models. Since models represent systems, and modeling tools can be used for forward and reverse engineering, the different proposals in the areas of *Visual Programming* and *Software Visualization* should be considered [112].

Many of the aspects that characterize Software Visualization techniques [113] have been exploited. Formal aspects such as **color**, **animation** and **multiple views** are used in the tool. **Color** is used to emphasize –e.g., to indicate the number of nested elements contained in a feature–, **animation** is used as a means for guiding the user –e.g., when layout changes–, and multiple views of models are offered –e.g., the foldable-node editor complements the diagrammatic one–. According to Price criteria, the visualization provided by the tool can be classified as **customizable**, since many aspects can be changed by users to adapt the tool to their needs. Support for interaction aspects such as **navigation** –e.g, moving from/to nested features in our editor– and **elision** –hiding content, as it happens when switching from different notations–, is also provided.

Improving interaction was one of the main goals of the present work. More visualization aspects related to interaction than the ones considered by Price are considered in the present work. For example, Spence in [111] stresses the relevance of **rearrangement of data** an aspect specially considered for the layout of our

editor.

Despite the popularity of feature model and the existence of tools to manage these model, tools have partially deal with the problem of handling complexity in feature models.

FMP [110] is an open source Eclipse plug-in which supports feature modeling and configuration. To represent feature models, FMP uses a hierarchical tree where each feature is represented by an icon followed of a text (see right side of Figure A.5). However, FMP does no support other organization criteria and it does not provide specific visualization techniques to handle complexity of feature models.

Pure::Variant [114] (P::V) is a commercial Eclipse plug-in which extends the Eclipse IDE to support the development and deployment of software product lines. To represent feature models, P::V uses a tree-view rendering similar to the one utilized on FMP (see left side of Figure A.1). Like FMP, P::V does not incorporate visualization techniques to handle model complexity.

Although not directly based on feature modeling, GEARS [115] is another commercial tool for modeling and configuring software variants. GEARS models variability as sets of parameters, where different parameter types stand for different kinds of variability. Although the parameters are not arranged into hierarchies as in feature diagrams, they can be organized into separate modules related by import statements.

Feature Designer [116] is a Visual Studio Integration Package, whose primary goal is to support FODA-consistent feature modeling for the .NET platform. Feature Designer supports cardinalities, specifying concrete configurations, feature composition and code generation. Feature Designer is implemented as a Domain-Specific Language in Visual Studio DSL Tools. As represented in the right side of Figure A.1, Feature designer incorporates a graphical 2D tree editor. However, the editor does not support customizing the notation style dynamically as MFM does. Moreover, Feature Designer does not incorporate techniques to deal with the graphic representation of complex models.

A.5 Conclusions and Future Work

In this appendix, we have proposed some visualization techniques to make feature editors scale as the model grows, considering also user customization. We have analyzed the particular visualization needs demanded by Feature Modeling to detect some points of improvement. Finally we have developed the MFM editor that addresses the detected limitations offering editing capabilities suitable for the feature modeling of large systems. In addition, MFM has been integrated in the Moskitt tool set for the support of software development process.

In our future work, we are particularly interested in feature constraints. We plan to work on visualization techniques to graphically express contrails persevering the readability, specially on complex feature models.

At http://www.pros.upv.es/labs/projects/mfm the source and binary distribution of Moskitt Feature Modeler are available. Furthermore, some screencasts and the MFM-FMP transformation are also available.