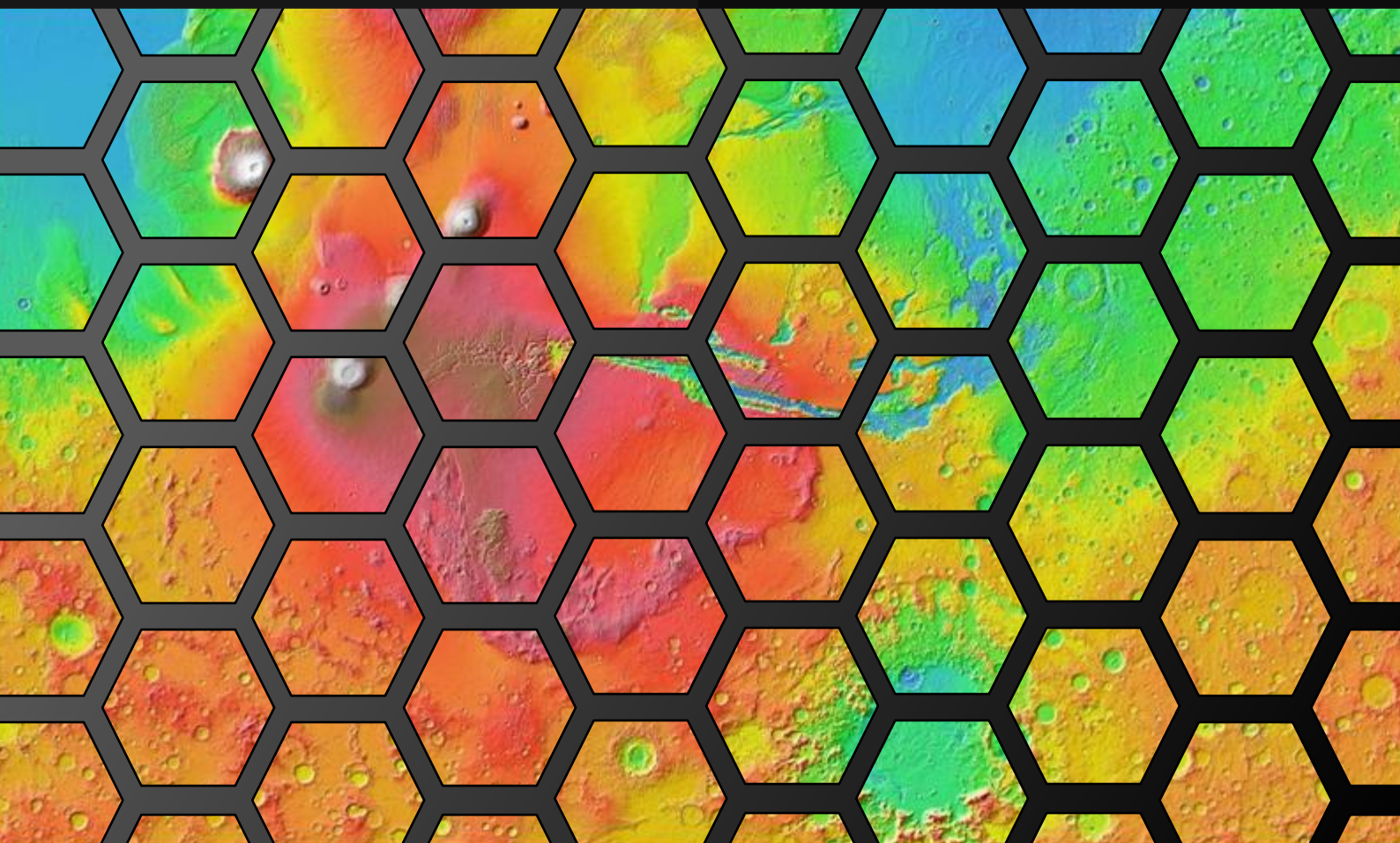


*A thesis submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science*

Achieving Autonomic Computing through the Use of Variability Models at Run-time

Carlos Cetina Englada



UNIVERSIDAD
POLITECNICA
DE VALENCIA

Thesis advisors:
Dr. Vicente Pelechano Ferragud
Dr. Joan Fons i Cors

Supervisors:

Dr. Vicente Pelechano Ferragud (Universidad Politécnica de Valencia)

Dr. Joan Fons i Cors (Universidad Politécnica de Valencia)

Dissertation Committee:

Prof. Dr. Óscar Pastor López (Universidad Politécnica de Valencia)

Dr. Pedro José Valderas Aranda (Universidad Politécnica de Valencia)

Dr. Antonio Ruiz Cortés (Universidad de Sevilla)

Prof. Dr. Antonio Vallecillo Moreno (Universidad de Málaga)

Prof. Dr. Jesús Joaquín García Molina (Universidad de Murcia)

Classification (ACM 1998):

Categories and subject descriptors: D.2.13 [Software Engineering]: Reusable Software: Domain Engineering; D.2.1 [Software Engineering].

Requirement/Specifications: Languages, Tools.

General Terms: Design.

Additional Key Words and Phrases: Software Product Lines, Model Driven Development, Autonomic Computing.

ABSTRACT

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. Adaptability is emerging as a necessary underlying capability, particularly for highly dynamic systems such as context-aware or ubiquitous systems. These systems have reached a level of complexity where the human effort required to get the systems up and running and keeping them operational is getting out of hand.

By automating tasks such as installation, adaptation, or healing, Autonomic Computing envisions computing environments that evolve without the need for human intervention. Even though there is a fair amount of work on architectures and their theoretical design, Autonomic Computing was criticised as being a “hype topic” because very little of it has been fully implemented. Furthermore, given that the autonomic system must change states at runtime and that some of those states may emerge and are much less deterministic, there is a great challenge to provide new guidelines, techniques and tools to help autonomic system development.

This thesis shows that building up on the central ideas of Model Driven Development (Models as first-order citizens) and Software Product Lines (Variability Management) can play a significant role as we move towards implementing the key self-management properties associated with autonomic computing. The presented approach encompass systems that are capable of modifying their own behavior with respect to changes in their operating environment, by using variability models as if they were the policies that drive the system’s autonomic reconfiguration at runtime. Under a set of reconfiguration commands, the components that make up the architecture dynamically cooperate to change the configuration of the architecture to a new configuration.

This work also provides the implementation of a Model-Based Reconfiguration Engine (MoRE) to blend the above ideas. Given a context event, MoRE queries the variability models to determine how the system should evolve, and then it provides the mechanisms for modifying the system architecture accordingly. The presented work has been validated from three different perspectives: (1) Scalability of the approach, (2) reliability-based risk of run-time reconfigurations and (3) degree of autonomic behavior achieved. This evaluation was performed with the participation of human subjects by means of a Smart Hotel case study which was deployed with real devices.

Experimentation shows that our approach achieves satisfactory results with regard to scalability and reliability-based risk; nevertheless, we found some scenarios which required a greater level of detail to define the autonomic behaviour since these scenarios deal more directly with user preferences and tastes. However, even though this lack of coverage could be complemented by the development of specific components for the unsupported cases, it does not seem economically realistic to build individual features to suit each user. Our intent is to focus on commonalities and abstractions that are valid across a set of users, looking for a trade-off between personalization and reusability.

RESUMEN

Cada vez más los sistemas software necesitan adaptar su comportamiento dinámicamente como respuesta a eventos de su propia infraestructura o del entorno físico que los rodea. La adaptabilidad se está convirtiendo en una capacidad básica para los sistemas software, particularmente para los sistemas altamente dinámicos como es el caso de los sistemas sensibles al contexto o los sistemas ubicuos. Sin embargo, estos sistemas han alcanzado un nivel de complejidad donde el esfuerzo requerido para mantenerlos operativos es demasiado elevado.

Mediante la automatización de tareas como la instalación, adaptación o reparación, la Computación Autónoma propone entornos de computación que evolucionan sin la necesidad de intervención por parte de los usuarios. Sin embargo, pese a que existe una razonable cantidad de trabajo en el ámbito de su diseño teórico, la Computación Autónoma ha sido criticada como un “tema demasiado ambicioso” debido a la falta de implementaciones que materialicen las ideas propuestas. Además, la naturaleza de los sistemas de computación autónoma (donde su estado cambia en tiempo de ejecución de una manera dinámica) plantea un gran desafío para dar soporte al desarrollo de los mismos mediante guías, técnicas y herramientas.

Esta tesis propone que la combinación de las ideas principales del Desarrollo de Software Dirigido por Modelos (los modelos como artefactos de primer orden) y las Líneas de Producto Software (la gestión de la variabilidad) puede jugar un papel importante para implementar las propiedades de autogestión propuestas por la Computación Autónoma. La propuesta presentada en esta tesis desarrolla sistemas que son capaces de modificar su propio comportamiento de acuerdo a cambios en su entorno. Esto se consigue utilizando modelos de variabilidad que juegan el rol de las políticas que dirigen la reconfiguración autónoma del sistema en tiempo de ejecución. Bajo un conjunto de comandos de reconfiguración, los componentes que forman la

arquitectura software cooperan dinámicamente para cambiar la configuración del sistema.

Este trabajo también proporciona la implementación de un Motor de Reconfiguración basado en Modelos (MoRE por sus siglas en inglés) para materializar las ideas propuestas. Dado un evento de contexto, MoRE consulta los modelos de variabilidad para determinar cómo debe evolucionar el sistema y luego proporciona mecanismos para modificar la arquitectura del sistema en consecuencia. El trabajo presentado ha sido validado desde tres perspectivas diferentes: (1) la escalabilidad de la propuesta, (2) los riesgos basados en la seguridad de las reconfiguraciones y (3) el nivel de comportamiento autónomo conseguido. Esta evaluación fue realizada con la participación de usuarios mediante el caso de estudio de un Hotel Inteligente que utilizaba dispositivos reales.

Los experimentos muestran que la propuesta obtiene resultados satisfactorios respecto a escalabilidad y riesgo de la reconfiguraciones; sin embargo, encontramos algunos escenarios que requirieron un mayor nivel de detalle para definir el comportamiento autónomo, ya que estos escenarios entraban en conflicto con preferencias de los usuarios. Aunque estos escenarios podían ser abordados mediante componentes específicos para los casos no soportados, no parece realista construir características del sistema para satisfacer a cada usuario de forma individual. Nuestra intención es centrarnos en generalizaciones y abstracciones que sean válidas a lo largo de conjuntos de usuarios, buscando un equilibrio entre personalización y reutilización.

RESUM

Cada vegada més els sistemes de programari necessiten adaptar el seu comportament dinàmicament com a resposta a esdeveniments de la seua pròpia infraestructura o de l'entorn físic que els envolta. L'adaptabilitat s'està convertint en una capacitat bàsica per als sistemes de programari, particularment per a aquells sistemes altament dinàmics, com és el cas dels sistemes sensibles al context o els sistemes ubicus. No obstant, aquests sistemes han arribat a un nivell de complexitat on l'esforç requerit per a mantindre-los operatius es massa elevat.

Mitjançant l'automatització de tasques com la instal·lació, l'adaptació o la reparació, la Computació Autònoma proposa entorns de computació que evolucionen sense la necessitat d'intervenció per part dels usuaris. No obstant, malgrat que existeix una raonable quantitat de treball en disseny teòric d'arquitectures, la Computació Autònoma ha estat criticada com “un tema massa ambiciós” a causa de la falta d'implementacions que materialitzen les idees proposades. A més, la naturalesa dels sistemes de Computació Autònoma (on el seu estat canvia en temps d'execució d'una manera dinàmica) planteja un gran desafiament per a suportar el desenvolupament d'aquest tipus de sistemes mitjançant guies, tècniques i eines.

Aquesta tesi mostra que la combinació de les idees principals del Desenvolupament de Programari Dirigit per Models (models com artefactes de primer ordre) i les Línies de Producte de Programari (gestió de la variabilitat) pot jugar un paper important cap a la implementació de les propietats d'autogestió proposades per la Computació Autònoma. La proposta presentada proposa sistemes que són capaços de modificar el seu propi comportament d'acord a canvis en el seu entorn. Açò s'aconsegueix emprant models de variabilitat com si fossen les polítiques que dirigeixen la reconfiguració autònoma del sistema en temps d'execució. Sota un conjunt de comandaments de reconfiguració, els components que formen l'arquitectura

de programari cooperen dinàmicament per a permetre canviar el sistema d'una configuració a una altra.

Aquest treball també proporciona la implementació d'un Motor de Reconfiguracions basat en Models (MoRe per les seues sigles en anglés) que materialitza les idees proposades. Donat un esdeveniment de context, MoRe consulta els models de variabilitat per a determinar com ha d'evolucionar el sistema i després proporciona mecanismes per a modificar l'arquitectura del sistema en conseqüència. El treball presentat ha estat validat des de tres perspectives diferents: (1) escalabilitat de la proposta, (2) riscos basats en la seguretat de les reconfiguracions i (3) nivell de comportament autònom aconseguit. Aquesta avaluació va ser realitzada amb la participació d'usuaris mitjançant el cas d'estudi d'un Hotel Intel·ligent que utilitzava dispositius reals.

Els experiments mostren que la proposta obté resultats satisfactoris respecte a escalabilitat i risc de les reconfiguracions; no obstant això, vam trobar alguns escenaris que van requerir un major nivell de detall per a definir el comportament autònom, ja que aquests escenaris entraven en conflicte amb algunes preferències dels usuaris. Encara que aquests escenaris podien ser abordats mitjançant components específics per als casos no suportats, no sembla realista construir característiques del sistema per a satisfer a cada usuari de forma individual. La nostra intenció és centrar-nos en generalitzacions i abstraccions que siguen vàlides per al conjunt total d'usuaris, cercant un equilibri entre personalització i reutilització.

CONTENTS

1	Introduction	2
1.1	Overview of the Chapter	2
1.2	Motivation	3
1.3	Problem Statement	7
1.4	Contribution	8
1.5	Research Methodology	10
1.6	Thesis Context	11
1.7	Outline	12
2	Background	16
2.1	Overview of the Chapter	16
2.2	Autonomic Computing	17
2.2.1	Definition	17
2.2.2	Properties of Autonomic Computing	18
2.2.3	The MAPE-K Autonomic Loop	20
2.3	Model Driven Development	26
2.3.1	Definition	27
2.3.2	Model Driven Software Development Initiatives	27
2.3.3	Domain Specific Languages	29
2.4	Software Product Lines	31
2.4.1	Definition	31
2.4.2	Software Product Line Processes	32
2.4.3	Dynamic Software Product Lines	34
2.5	Conclusions	35

3	State of the Art	38
3.1	Overview of the Chapter	38
3.2	Classification Criteria	39
3.2.1	Adoption Level of Autonomic Computing	39
3.2.2	Relevance of the Autonomic Computing	40
3.2.3	Reinforcement of the Autonomic Knowledge	41
3.2.4	Maturity of the Software Engineering Approach	41
3.3	Analysis of Approaches for System Family Reconfiguration	44
3.3.1	Gomaa and Hussein Approach	45
3.3.2	Lee and Kang Approach	47
3.3.3	Hallsteinsen et al. (MADAM) Approach	50
3.3.4	White et al. Approach	52
3.3.5	Trinidad et al. Approach	55
3.3.6	Mori et al. Approach	57
3.3.7	Hallsteinsen et al. (MUSIC) Approach	59
3.3.8	Parra et al. Approach	62
3.3.9	Istoan et al. Approach	64
3.4	Discussion	66
3.4.1	Architectural Patterns	68
3.5	Conclusions	71
4	Overview of the Approach	74
4.1	Overview of the Chapter	74
4.2	Introduction	75
4.3	Main Building Blocks	78
4.4	Application	80
4.5	Implementation	85
4.6	Validation	86
4.7	Conclusions	88
5	Autonomic Computing through the Use of Variability Models	90
5.1	Overview of the Chapter	90

5.2	Variability Modelling	91
5.2.1	Variability and Software Product Lines	91
5.2.2	Model Driven Software Product Lines	93
5.3	Specifying Reconfigurations through Feature Models	100
5.3.1	Evaluation of the Autonomic Behaviour through Feature Models.	103
5.4	Model-Based Validation of Reconfigurations	104
5.5	Applying the approach to other Variability Language: Autonomic Behaviour through Common Variability Language	106
5.6	Conclusions	110
6	Achieving Autonomic Computing through Models at run-time	112
6.1	Overview of the Chapter	112
6.2	Renconfigurable System Architectures	113
6.2.1	The OSGi Framework: A Realization of the Renconfigurable System Architecture	116
6.3	Reconfiguring the System Architecture through Feature Models	119
6.4	MoRE: Model-based Reconfiguration Engine	123
6.4.1	MoRE Model Operations	124
6.4.2	MoRE Reconfiguration Actions	131
6.5	Scalability Evaluation of Model-management Technologies at Run-time	138
6.6	Conclusions	141
7	Strategies for Variability Transformation at run-time	144
7.1	Overview of the Chapter	144
7.2	From Design Variability to run-time Variability: Challenges	145
7.3	Managing Variability at Run-time	148
7.3.1	Feature-based Approach	148
7.3.2	Fragment-based Approach	150
7.3.3	Assessment between Feature-based and Fragment-based	152
7.4	Strategies for Variability Transformation	154
7.4.1	Common Operations of the Strategies	154

7.4.2	Regenerative Strategy	157
7.4.3	Incremental - Copy Strategy	159
7.4.4	Incremental - Move Strategy	160
7.4.5	Implementation of the Strategies	161
7.5	Validating the Strategies Implementation	162
7.5.1	Tool Support for Testing Strategies	164
7.6	Extra-Functional Properties of Strategies	166
7.7	Applying the Strategies to Smart Homes	168
7.8	Conclusions	169
8	Evaluation of the Proposal	172
8.1	Overview of the Chapter	172
8.2	Background on DSPL evaluation	174
8.3	The Smart Hotel Case Study	175
8.3.1	Reconfiguration Scenarios of the Smart Hotel	176
8.4	Evaluation Logistics of the Case Study	179
8.4.1	Participants and Training	179
8.4.2	Challenges to involve Human participants in DSPL Evaluation	180
8.4.3	Experiment Operation	184
8.4.4	Data Collection	186
8.4.5	Keeping Track of the Reconfigurations	186
8.5	Evaluation	189
8.6	Discussion	191
8.6.1	Introducing User Confirmations to Reconfigurations	192
8.6.2	Improving Reconfiguration Feedback	194
8.6.3	Introducing Rollback Capabilities to Reconfigurations	195
8.7	Conclusions	196
9	Conclusions and Future Work	198
9.1	Overview of the Chapter	198
9.2	Contributions	199
9.3	Research Visits	202

9.4	Assessment and Future Work	203
9.4.1	Enabling End-user participation in the Design of Reconfigurable Systems	203
9.4.2	Enhancing Run-time Reconfigurations to Take into Account End-user Preferences	204
9.4.3	Providing Metrics to Quantify System Reconfiguration Capabilities	205
9.4.4	Guarantying Quality Properties on Run-time Reconfigurations	205
9.4.5	Addressing other Application Domains	206
9.5	Publications	207
9.6	Senior Theses Codirected	211
9.7	Seminars	212
9.8	Final Conclusion	213
	Appendix	214
A	The Smart Hotel	
	Case Study	214
A.1	Overview of the Case Study	215
A.2	Scenarios of the Smart Hotel	217
A.3	Functionality of the Smart Hotel	220
A.4	Software Architecture of the Smart Hotel	221
A.5	Reconfigurations in the Smart Hotel	223
A.5.1	Check-in	224
A.5.2	Entering the room	229
A.5.3	Working	232
A.5.4	Watching a Movie	235
A.5.5	Sleeping	238
A.5.6	Leaving the room	241
A.5.7	House Keeping	244
A.5.8	Check-out	247
A.6	Summary	250

B Tool Support	254
B.1 Support for Designing Autonomic Behaviour	254
B.1.1 Support for Reconfiguration Analysis	258
B.2 Support for Model-based Run-time Reconfigurations	259
 Bibliography	 264

LIST OF FIGURES

1.1	Scope of Chapter 1	2
1.2	Research methodology followed in this thesis.	11
1.3	Roadmap of this Thesis.	12
2.1	Scope of Chapter 2	16
2.2	IBM's MAPE-K reference model for autonomic control loops	20
3.1	Scope of Chapter 3	38
3.2	Gomaa and Hussein Approach	46
3.3	Lee and Kang Approach	48
3.4	Hallsteinsen et al. Approach	51
3.5	White et al. Approach	53
3.6	Trinidad et al. Approach	56
3.7	Mori et al. Approach	58
3.8	Hallsteinsen et al. Approach	61
3.9	Parra et al. Approach	63
3.10	Istoan et al. Approach	65
3.11	Classification of the Approaches: Maturity - Autonomic Level	67
3.12	Connected DSPL Overview	69
3.13	Disconnected DSPL Overview	70
3.14	Classification of DSPL: Scope - Infrastructure	72
4.1	Scope of Chapter 4	74
4.2	Main Building Blocks of the Approach	78
4.3	Simplified overview of the Process to Apply the Approach	81
4.4	Overview of the Process to Apply the Approach	82

4.5	Overview of the Run-time Reconfiguration of the Approach	85
5.1	Scope of Chapter 5	90
5.2	SPL main concepts	92
5.3	SPL following the MDD Approach	94
5.4	Feature model of a smart home.	96
5.5	Main concepts of PervML Pervasive Systems	97
5.6	Two configurations of a Pervsive System.	98
5.7	Snapshot of a Feature Model.	99
5.8	OWL Ontology for Autonomic Homes.	101
5.9	Visualizing variability as an adaptation space.	104
5.10	Base-Variation-Resolution Approach.	107
5.11	Modelling Variability with CVL.	108
5.12	Applying CVL for Autonomic Homes.	109
5.13	Feature Modelling and CVL.	110
6.1	Scope of Chapter 6	112
6.2	Reconfiguration Pattern of a DSPL Architecture.	114
6.3	The Smart Home from an OSGi perspective.	118
6.4	Overview of the model-based reconfiguration process.	120
6.5	Architecture Increment and Decrement given a Resolution.	122
6.6	The model-based reconfiguration process overview.	123
6.7	Calculating $A(A \triangle \nabla)$ through the Model Operations.	130
6.8	Experimental results.	140
7.1	Scope of Chapter 7	144
7.2	Managing Variability.	148
7.3	Overview of Feature-based superimposition.	149
7.4	Overview of the Variability Transformation.	151
7.5	Common operations for fragment substitution.	154
7.6	Regenerative Strategy.	158
7.7	Incremental-Copy Strategy.	159
7.8	Incremental-Move Strategy.	160

7.9	Possibility Space.	163
7.10	Testing Tool for Strategies.	165
8.1	Scope of Chapter 8	172
8.2	Reconfigurations among Scenarios.	178
8.3	Context Cards for triggering DSPL reconfigurations.	182
8.4	Visualizing reconfiguration effects by means of the Configuration Viewer.	183
8.5	Experimentation set-up.	185
8.6	Information of the Reconfigurations is Stored as Model-based Traces.	187
8.7	Snapshot of MoRE Traces tool.	188
8.8	Overall results from the Case Study.	190
8.9	Categories for confirmation of reconfigurations.	193
9.1	Scope of Chapter 9	198
A.1	Hotel's Smart Room	217
A.2	Reconfiguration through the Smart Hotel Scenarios	219
A.3	Feature Model of the Smart Hotel	221
A.4	PervML Model of the Smart Hotel	222
A.5	Mapping between features and PervML	223
A.6	Another mapping between features and PervML	224
A.7	Feature model of the Check-in Scenario	225
A.8	PervML model of the Check-in Scenario	226
A.9	Feature model of the Entering the room Scenario	230
A.10	PervML model of the Entering the Room Scenario	231
A.11	Feature model of the Working Scenario	233
A.12	PervML model of the Working Scenario	234
A.13	Feature model of the Working Scenario	236
A.14	PervML model of the Working Scenario	237
A.15	Feature model of the Working Scenario	239
A.16	PervML model of the Working Scenario	240
A.17	Feature model of the Leaving the Room	242
A.18	PervML model of the Leaving the room	243

A.19 Feature model of the Housekeeping Scenario	245
A.20 PervML model of the Housekeeping Scenario	246
A.21 Feature model of the Check-out Scenario	248
A.22 PervML model of the Check-out Scenario	249
B.1 Tool Support for Design Time.	255
B.2 Screenshots of the Tool Support for Design Time	256
B.3 Screenshot of the Resolution Editor	258
B.4 Screenshot of the Reconfiguration Analysis Tool	259
B.5 MoRE implemented as OSGi Bundles.	260
B.6 Package Diagram of MoRE Implementation	261
B.7 Class Diagram of MoRE Implementation	263

LIST OF TABLES

3.1	Template for Approach Classification.	44
3.2	Classification of Gomaa and Hussein Approach.	47
3.3	Classification of Lee and Kang Approach.	49
3.4	Classification of Hallsteinsen et al. Approach.	52
3.5	Classification of White et al. Approach.	54
3.6	Classification of Trinidad et al. Approach.	57
3.7	Classification of Mori et al. Approach.	59
3.8	Classification of Hallsteinsen et al. Approach.	62
3.9	Classification of Parra et al. Approach.	64
3.10	Classification of Istoan et al. Approach.	66
7.1	Extra-Functional Properties of the Strategies	166
8.1	Scale Definition of Reliability Metrics.	180
A.1	Reconfiguration Table: Check-in Scenario.	228
A.2	Reconfiguration Table: Entering the Room.	232
A.3	Reconfiguration Table: Working.	235
A.4	Reconfiguration Table: Watching a Movie.	238
A.5	Reconfiguration Table: Sleeping.	241
A.6	Reconfiguration Table: Leaving the Room.	244
A.7	Reconfiguration Table: Housekeeping.	247
A.8	Reconfiguration Table: Check-out.	250

Chapter 1. INTRODUCTION

“The real voyage of discovery consists not in seeking new lands but seeing with new eyes.”

– Marcel Proust (1871-1922).

1.1 Overview of the Chapter

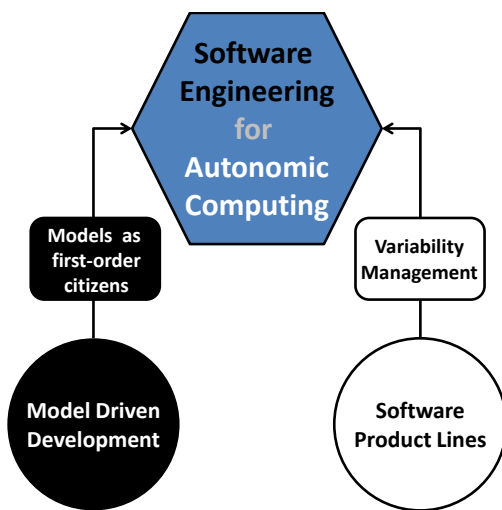


Figure 1.1: Scope of Chapter 1

This thesis brings together the fields of Software Product Lines and Model Driven Development with the purpose of addressing the creation of autonomic computing systems. A system with autonomic capabilities installs, configures, tunes, and maintains its own components at run-time, envisioning computing environments that evolve without the need for human intervention. However, there is a great challenge to provide implementations of current theoretical designs as well as support for autonomic system development [1].

The contribution of this work is not only an execution platform but also techniques and tools to support autonomic system engineers from system design to execution. At design time, we provide variability modelling techniques to **design** and **validate** the autonomic behaviour. At run-time, we provide an enhanced model-based **implementation** of the reference model for autonomic control [2], which also enables a **posteriori analysis** of the overall running of the system by means of both debugging and execution traces capabilities.

In this work, autonomic computing is achieved by leveraging variability models at run-time. In this way, the modelling effort made at design time provides a richer semantic base for autonomic behavior during execution. Variability models specify the possible configurations of the system, while a reconfigurable architecture can be rapidly retargeted to a specific configuration.

In order to support the proposal, a Model-based Reconfiguration Engine (named MoRE) was developed. In response to context conditions, MoRE uses at run-time the variability models from design time to determine how the system should move from a consistent architecture to another consistent architecture by means of reconfiguration strategies. Given the fact that these strategies provide different extra-functional properties (such as different performances), we also provide a catalog of these strategies in order to enable engineers to set up MoRE with the most suitable strategy for each particular concern such as debugging or performance.

The rest of this chapter is organized as follows. First, we introduce the motivation of this thesis. Then, the problem that this work resolves is stated in detail. Next, the main contributions of this thesis are summarized. The research methodology that we have followed is also presented. Finally we explain the context in which the work of this thesis has been performed, and we present the outline of the thesis.

1.2 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 [3]. Adaptability is emerging as a necessary underlying capability, particularly for highly dynamic systems such as context-aware [4, 5] or ubiquitous [6, 7] systems. These systems have reached a level of complexity where the human effort required to get the systems up and running and keeping them operational is getting out of hand. With more and more digital services being added to our surroundings, simplicity is highly appreciated by users, as stated in [8, 9].

Autonomic computing [10] envisions computing environments that evolve with-

out the need for human intervention. A system with autonomic capabilities installs, configures, tunes, and maintains its own components at run-time. The term “autonomic” comes from biology. In the human body, the autonomic nervous system takes care of unconscious reflexes, i.e. body functions that do not require our attention such as the size of the pupil, the digestive functions, the rate and depth of respiration and dilatation or constriction of the blood vessels. Without the autonomic nervous system, we would be constantly busy consciously adapting our body to its needs and to the environment.

Inspired by biology, autonomic computing has evolved as a discipline to create software systems and applications that self-manage in a bid to overcome the complexities and inability to maintain current and emerging systems effectively. To this end autonomic endeavours cover the broad span of computing from end-to-end applications to infrastructure middlewares, and it is already demonstrating its feasibility and value by automating tasks such as installation [11], healing [12], and updating [13].

However, although there is a fair amount of work on architectures and their theoretical design, very little of it has been fully implemented, which is currently one of the main challenges of Autonomic Computing as stated in a recent survey [1] (*Challenge 1*). Furthermore, defining appropriate abstractions and models for understanding, controlling, and designing autonomic behavior is another important challenge at the heart of AC [10] (*Challenge 2*). Another challenge facing this community lies in the ability to carry out robust software engineering to provide solidly built autonomic systems [14] (*Challenge 3*).

Current software engineering practice defines a system in a more or less pre-implementation state where requirements have been agreed a priori. Given that the autonomic system must change states at run-time and that some of those states may emerge and are much less deterministic, there is a great challenge to provide new guidelines, techniques and tools to help autonomic system development.

Consequently, autonomic computing needs software engineering approaches that better handle abstraction while being suitable in their ability to represent dynamics in a ever-changing system. To this end, we suggest that the combination of

both Model Driven Development [15] and Software Product Lines [16] can lead to a systematic software engineering approach for the development of such systems.

- **Model Driven Development** is a paradigm capturing every important aspect of a software system through appropriate models. These models are not just auxiliary documentation artifacts; rather, they are source artifacts and can be used for automated analysis and/or code generation.

The use of model-driven techniques can contribute to realize the vision of autonomic computing, since models can be used as the autonomic knowledge of the system in order to provide a richer semantic base for run-time decision-making related to system adaptation [17].

- **Software Product Lines** shift from the development of an individual system to the development of reusable assets that are used to develop a family of systems. Variability management is the fundamental principle of Software Product Lines, which involves separating the product line into three parts—common components, parts common to some but not all products, and individual products with their own specific requirements—and managing these throughout development.

The use of Software Product Lines techniques can also contribute to realize the vision of autonomic computing. Variation points can be bind at run-time, initially when software is launched to adapt to the current environment, as well as during operation to adapt to changes in the environment [18].

On the one hand, Model Driven Development can contribute to address *Challenge 2* by appropriate model abstractions to represent the important aspects of the autonomic behaviour. On the other hand, Software Product Lines (specially Dynamic Software Product Lines) can contribute to address *Challenge 3* by handling the dynamicity of autonomic systems in a systematic manner. Finally, both Model Driven Development and Software Product Line communities have been highly productive with several tools now entering the commercialisation phase which can also contribute to *Challenge 1* by applying the former tools to produce Autonomic Computing implementations.

Building on the central ideas of Model Driven Development and Software Product Lines can play a significant role as we move towards implementing the key self-management properties associated with autonomic computing. Our research shows that autonomic behavior can be achieved by leveraging variability models at run-time. In this way, the modelling effort made at design time is not only useful for producing the system but also provides a richer semantic base for autonomic behavior during execution. The use of variability models at run-time brings new opportunities for autonomic capabilities as follows.

- **Use of model driven development techniques to control the autonomic behaviour.** The knowledge previously captured in variability models can be used to describe the variants in which a system can evolve. In response to changes in the context, the system itself can query these models to determine the necessary modifications to its architecture.
- **Use of Product Line architectures to support the autonomic behaviour.** Variation points and dynamic binding enables the creation of software architectures that can be rapidly retargeted to a specific configuration. When the system enters a particular context that requires adaptation, the product line architecture allows an easy reconfiguration since architecture components can dynamically appear or disappear from configurations, and communication channels can be established dynamically between the components.

The combination of the above ideas give birth to a Model-Based Reconfiguration Engine (MoRE). Given a context event¹, this engine can query the variability models to determine how the system should evolve, and it also provides the mechanisms for modifying the system architecture accordingly. Thus, MoRE-enabled systems can use the knowledge captured by variability models to drive its own autonomic evolution at run-time.

The smart home domain is a candidate to validate the above approach. This domain is suited for variability modelling techniques because of the high degree

¹We understand context event as any observable property by sensors that can impact system execution, e.g. end-user input, hardware devices, network connection properties.

of similarities among different systems; also, autonomic computing capabilities can address some of the domain's limitations such as minimal support for evolution as new technologies emerge or as an application type matures [19]. A planned reutilization of the modelling efforts invested at design time by MoRE can contribute to alleviate the former limitations of smart homes.

1.3 Problem Statement

The development of Autonomic Computing Systems is not a closed research topic. We can see from the above discussion how some problems still need to be considered. The work that is presented in this thesis help to improve the development of Autonomic Systems by addressing the *Challenges* presented above. In particular, the problems that this thesis addresses can be stated by means of the following problem statements.

- **Research Question 1.** How to carry out a software engineering approach for the development of autonomic systems in order to provide not only an execution platform but also techniques and tools to support engineers from system design to execution?
- **Research Question 2.** How model abstractions should be defined for controlling, and designing autonomic behavior as well as enabling the analysis of the autonomic behaviour before implementing it?
- **Research Question 3.** How to realize current theoretical design architectures for autonomic computing into executable implementations?

In conclusion, although autonomic computing has become increasingly interesting and popular it remains as a relatively immature topic from the point of view of Software Engineering. We believe that the research related to the above questions can contribute to push both researchers and practitioners towards a sound and seamless engineering support for autonomic computing.

1.4 Contribution

The main contributions of this thesis have been developed to answer the three research questions presented above. Next we summarize the main contributions of our research work.

1. The major contribution of this thesis is a **software engineering approach for autonomous computing** which combines the main ideas of Model Driven Development (models as first-order citizens) and Software Product Lines (variability management). This approach provides not only an execution platform but also techniques and tools to support autonomous system engineers from system design to execution.

On the one hand, we suggest the application of Scope, Commonality and Variability analysis [20] by means of variability modelling to specify the autonomous knowledge in terms of variants that are associated to context conditions. On the other hand, we demonstrate that the executing system can make use of the knowledge captured by variability models as if they were the policies that drive the autonomous behaviour of the system.

2. We show **how to design and validate the autonomous behaviour** by means of variability modelling techniques. Since the models that form the basis for reconfiguration strategies are available at design time, we are able to validate configurations in an early stage of the development process without first implementing them. Furthermore, we have automated this step using the analysis operations of the FAMA framework [21]. Specifically, we design the autonomous behaviour by means of modelling techniques as follows.

- (a) *Variability models* describe the system configurations and its variants.
- (b) *Domain specific languages* describe the system architecture.
- (c) *Weaving models* map system variants to software architecture components.

- (d) *Ontologies for context modelling* connect context conditions to system variants.

We also show the feasibility of leverage *as is* the above models at run-time to drive the autonomic behaviour. That is, we keep the same model representation at run-time that is used at design time: the XML Metadata Interchange (XMI) standard. This avoids the definition of technological bridges, because the same technologies used at design time for manipulating XMI models can be applied at run-time. In particular, our approach queries and updates the models at run-time using the widespread tools of the Eclipse Modelling Project².

3. We provide **a model-based implementation of the reference model for autonomic control** [2]. The four phases of this reference model (Monitor, Analyse, Plan and Execute) have been implemented in our Model-based Reconfiguration Engine: MoRE. By means of the former phases and the above models at run-time, MoRE determines how a system should be reconfigured for a target operational context, and then it modifies the system architecture accordingly. MoRE features different strategies to implement the former reconfiguration. These strategies have different extra-functional properties in order to address particular concerns such as debugging or performance.

The presented approach encompass systems that are capable of modifying their own behavior with respect to changes in their operating environment by using run-time reconfigurations. The presented work has been validated from three different perspectives as follows.

- **Scalability of the approach.** Since model manipulation at run-time, is subject to the same efficiency requirements as the rest of the system, we have evaluated this approach from the point of view of efficiency achieving positive results.
- **Reliability-based risk of run-time reconfigurations.** A failure in the reconfigurations can directly impact the user experience since the reconfig-

²<http://www.eclipse.org/modeling/>

urations are performed when the system is already under the users control. Therefore, we also evaluated the reliability-based risk of run-time reconfigurations, specifically, the probability of malfunctioning (Availability) and the consequences of malfunctioning (Severity).

- **Degree of autonomic behavior achieved.** This evaluation was performed with the participation of human subjects by means of a Smart Hotel case study which was deployed with real devices.

Moreover, we successfully identified and addressed two challenges associated with the involvement of human subjects in reconfiguration evaluation: enabling participants to (1) trigger the run-time reconfigurations and to (2) understand the effects of the reconfigurations. The evaluation of the case study reveals positive results regarding both Availability and Severity. However, the participant feedback highlights issues with recovering from a failed reconfiguration or a reconfiguration triggered by mistake. To address these issues, we provide some guidelines learned in the case study. Finally, we conclude that our approach achieved satisfactory results with regard to reliability-based risk; nevertheless, system engineers must provide users with more control over the reconfigurations or the users will not be comfortable with the resulting autonomic behaviour.

1.5 Research Methodology

In order to perform the work of this thesis, we will apply a research project following the design methodology for performing research in information systems as described by [22] and [23]. Design research involves the analysis of the use and performance of designed artefacts to understand, explain and, very frequently, to improve on the behaviour of aspects of Information Systems [23].

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

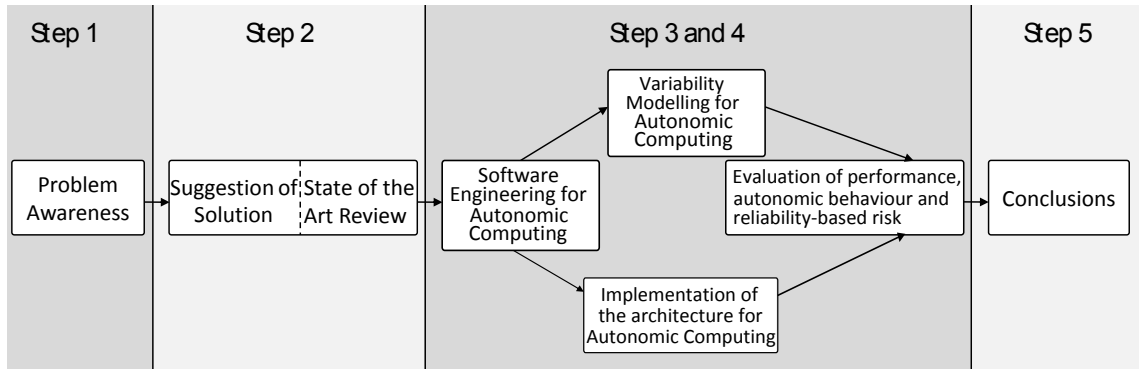


Figure 1.2: Research methodology followed in this thesis.

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

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

Finally, we will analyze the results of our research work in order to obtain several conclusions as well as to delimitate areas for further research (step 5).

1.6 Thesis Context

This thesis has been developed in the context of the *Research Center for Software Productions Methods* (Pros) of the *Technical University of Valencia*. The work that has made the development of this thesis possible is in the context of the following research projects.

- SESAMO: Construcción de Servicios Software a partir de Modelos. CYCIT project referenced as TIN2007-62894 (National Project).
- OSAMI Commons: Open Source Ambient Intelligence Commons. ITEA 2 project referenced as TSI-020400-2008-114 (European Project).

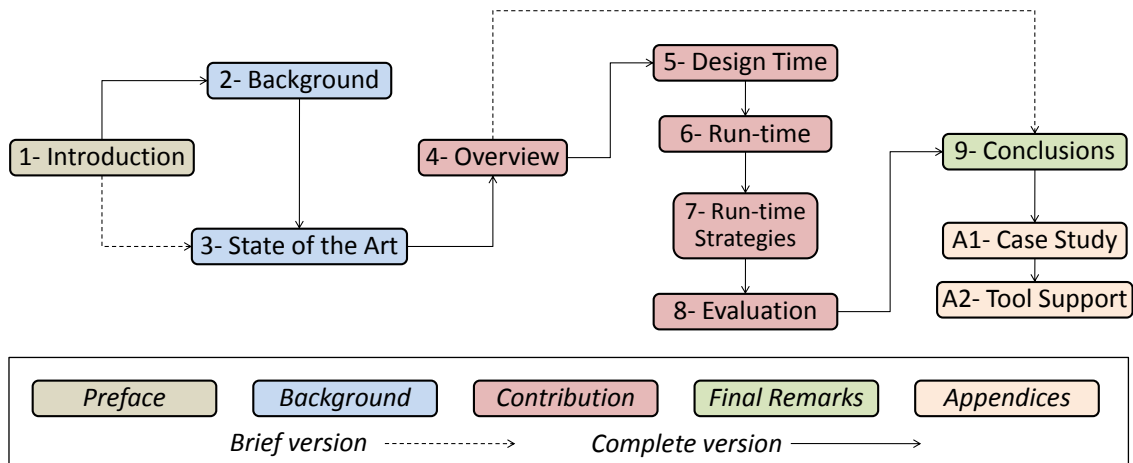


Figure 1.3: Roadmap of this Thesis.

Furthermore, this thesis builds up on other works that we have been developing during the last years. In particular, we have been applying model driven development to the Pervasive System domain. As a result, we introduced (1) a Domain Specific language for the specification of Smart Homes (PervML) [24], and (2) a tool (PervGT) [25] for the definition of PervML models and automatic code generation from these models to the final system implementation. For more information about this previous work, see <http://www.pros.upv.es/labs/projects/pervml>.

1.7 Outline

Figure 1.3 shows a roadmap for this thesis. It consists of nine chapters and two appendices as follows.

Chapter 2, *Background*. This Chapter presents the main concepts and characteristics of the approaches related with this thesis, in order to provide to the reader a basic background for understanding the overall thesis work. Specifically, this chapter presents autonomic computing, Model Driven Development and Software Product Lines.

Chapter 3, *State of the Art*. This chapter shows an analysis of the most important approaches that have been proposed to support run-time reconfiguration of system families. These approaches are classified according to criteria for evalu-

ating both the achieved autonomic behaviour and the methodology to achieve this behaviour.

Chapter 4, *Overview of the Approach.* This chapter introduces the present approach for the development of autonomic systems through the use of variability models at run-time. This overview covers the main building blocks of the approach as well as the process to apply it. In addition, the chapter also introduces how the approach has been evaluated throughout the case study of a Smart Hotel.

Chapter 5, *Autonomic Computing through the use of Variability Models.* This chapter argues how the knowledge captured in variability models is used for providing autonomic behaviour during execution. The chapter also shows how our approach is able to conduct a thorough analysis of the variability models for the purpose of validation.

Chapter 6, *Achieving Autonomic Computing Through Models at Run-time.* This chapter shows the model operations to query and update variability models at run-time in order to drive the reconfiguration of the architecture in response to context events. These variability models at run-time determine how a set of components can cooperate to change from one architecture configuration to another.

Chapter 7, *Strategies for Variability Transformation at Run-time.* This chapter presents different strategies (with different extra-functional properties) to implement the reconfiguration functionality provided by the model operations of Chapter 6. For example, MoRE can use an strategy with debugging support as long as the system is under development. When the development is finished and the system is going to be deployed, MoRE can use another strategy with better performance (but without debugging support).

Chapter 8, *Evaluation of the Proposal.* This chapter shows the evaluation of the proposal in terms of reliability-based risk of the run-time reconfigurations, which depends on both the probability that the reconfigurations will fail in the operational environment and the adversity of that failure.

Chapter 9, *Conclusions and Future Work.* This chapter presents the main contributions, results and publications of this work. In addition, this chapter discusses future research directions in connection to the limitations of the work.

Appendix A, *The Smart Hotel Case Study*. This appendix presents the case study of a Smart Hotel, which reconfigures its services according to changes in the surrounding context. This case study has been specifically developed to exercise reconfigurations that support the autonomic behaviour.

Appendix B, *Tool Support*. This appendix shows a general view of the tools proposed in this thesis to support the approach. These tools enable autonomic system engineers to specify the autonomic behaviour by means of variability models, and to take advantage of these specifications at run-time to drive the system evolution.

Chapter 2. BACKGROUND

“We shall not cease from exploration And the end of all our exploring Will be to arrive where we started And know the place for the first time. ”

– Thomas Stearns Eliot (1888-1956).

2.1 Overview of the Chapter

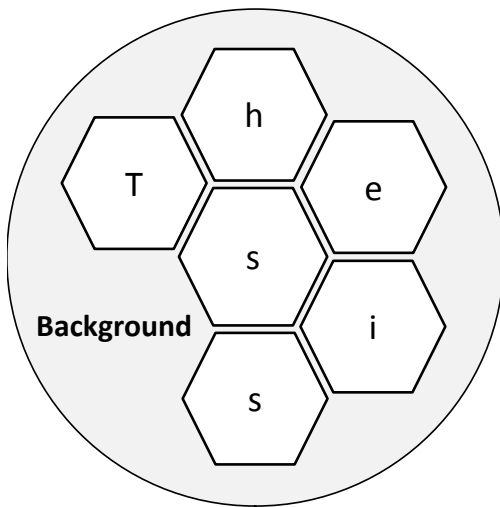


Figure 2.1: Scope of Chapter 2

In this chapter the background of the Thesis is introduced. The background in our case is conformed by the approaches that are related to the objective of this work: to achieve autonomous computing through the use of variability models at run-time. Therefore, this chapter presents the main concepts and characteristics of these approaches in order to provide a basic background for understanding the overall thesis work. Specifically, we present autonomous computing (target) and both model driven development and Software

Product Lines (means). These approaches are briefly introduced as follows.

First, we present **Autonomous Computing**, which 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.

Second, we present **Model Driven Development**, which 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.

Finally, we present **Software Product Lines** engineering, which intends to produce a set of products that share a common set of assets in an specific domain. These 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 a stand-alone systems to the development of a systems family.

2.2 Autonomic Computing

In October 2001, IBM released a manifesto [10] 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.

2.2.1 Definition

Inspired by biology, autonomic computing has evolved as a discipline to create software systems and applications that self-manage in a bid to overcome the complexities and inability to maintain current and emerging systems effectively. To this end autonomic endeavours cover the broad span of computing from end-to-end applications to infrastructure middlewares, and are already demonstrating their feasibility and value.

In 2001, IBM suggested the concept of autonomic computing. In their manifesto, complex computing systems are compared to the human body, which is a complex system, but has an autonomic nervous system that takes care of most bodily functions, thus removing from our consciousness the task of coordinating all our bodily functions. IBM suggested that complex computing systems should also have

autonomic properties, i.e. should be able to independently take care of the regular maintenance and optimization tasks, thus reducing the workload on the system administrators. IBM also distilled the four properties of a self-managing (i.e. autonomic) system: self-configuration, self-optimization, self-healing and self-protecting. As stated by Alan Ganek who is on behalf of Autonomic Computing in IBM:

“Autonomic computing is the ability of systems to be more self-managing. The term autonomic comes from the autonomic nervous system, which controls many organs and muscles in the human body. Usually, we are unaware of its workings because it functions in an involuntary, reflexive manner – for example, we don’t notice when our heart beats faster or our blood vessels change size in response to temperature, posture, food intake, stressful experiences and other changes to which we’re exposed. And, by the way, our autonomic nervous system is always working”

2.2.2 Properties of Autonomic Computing

The main properties of Autonomic Computing as portrayed by IBM are self-configuration, self-optimisation, self-healing and self-protection. Here is a brief description of these properties (for more information, see [26, 27]):

- **Self-configuration.** An autonomic computing system configures itself according to high-level goals, i.e. by specifying what is desired, not necessarily how to accomplish it. This can mean being able to install and set itself up based on the needs of the platform and the user.
- **Self-optimization.** An autonomic computing system optimises its use of resources. It may decide to initiate a change to the system proactively (as opposed to reactive behaviour) in an attempt to improve performance or quality of service.
- **Self-healing.** An autonomic computing system detects and diagnoses problems. The kinds of problems that are detected can be interpreted broadly: they can be as low-level as bit-errors in a memory chip (hardware failure) or

as high-level as an erroneous entry in a directory service (software problem) [28]. If possible, it should attempt to fix the problem, for example by switching to a redundant component or by downloading and installing software updates. However, it is important that as a result of the healing process the system is not further harmed, for example by the introduction of new bugs or the loss of vital system settings. Fault tolerance is an important aspect of self-healing. That is, an autonomic system is said to be reactive to failures or early signs of a possible failure.

- **Self-protection.** An autonomic system protects itself from malicious attacks but also from end users who inadvertently make software changes, e.g. by deleting an important file. The system autonomously tunes itself to achieve security, privacy and data protection. Security is an important aspect of self-protection, not just in software, but also in hardware. A system may also be able to anticipate security breaches and prevent them from occurring in the first place. Thus, the autonomic system exhibits proactive features.

The concepts behind the self-* properties were not entirely new to IBM's autonomic computing initiative. For example, a query optimiser, resource manager or routing software in Data Base Management Systems (DBMS), operating systems and networks, respectively, all allow those systems to self-manage. However the Self-Managing systems' community are coming to an agreement that the term autonomic computing is not being used to describe these systems but those in which the query plan, resource management or routing decision changes to reflect the current environmental context; reflecting dynamism in the system. That is, the DBMS query plan changes as the query is running.

In addition, other Adaptive systems have contained some elements of the above properties for some time, especially to provide self-optimisation. Early examples of this can be seen in streaming media systems where the codec of the stream changes with network bandwidth fluctuations, the goal being to keep music or video playback as high a quality as possible, e.g. Kendra [29] and Real Surestream [30]. However the autonomic community is more and more identifying a system as autonomic if it

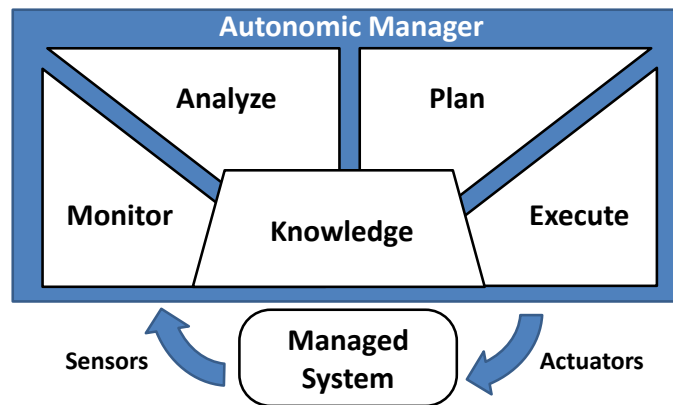


Figure 2.2: IBM's MAPE-K reference model for autonomic control loops

exhibits more than one of the self-management properties described earlier [31].

2.2.3 The MAPE-K Autonomic Loop

To achieve autonomic computing, IBM has suggested a reference model for autonomic control loops [2], which is sometimes called the MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) loop and is depicted in Figure 2.2. This model is being used more and more to communicate the architectural aspects of autonomic systems.

The MAPE-K autonomic loop is similar to, and probably inspired by, the generic agent model proposed by Russel and Norvig [32], in which an intelligent agent perceives its environment through sensors, and uses these percepts to determine actions to execute on the environment.

In the MAPE-K autonomic loop, the managed element represents any software or hardware resource that is given autonomic behaviour by coupling it with an autonomic manager. Thus, the managed element can for example be a web server or database, a specific software component in an application (e.g. the query optimiser in a database), the operating system, a cluster of machines in a grid environment, a stack of hard drives, a wired or wireless network, a CPU, a printer, etc.

Sensors, often called probes or gauges, collect information about the managed element. For a web-server, that could include the response time to client requests, network and disk usage, CPU and memory utilisation. A considerable amount of research is involved in monitoring servers [33, 34, 35, 36, 37].

Actuators carry out changes to the managed element. The change can be coarse grained, e.g. adding or removing servers to a web server cluster [38], or fine-grained, e.g. changing configuration parameters in a web server [36, 39].

Autonomic manager

The data collected by the sensors allows the autonomic manager to monitor the managed element and execute changes through actuators. The autonomic manager is a software component that ideally can be configured by human administrators using high-level goals and uses the monitored data from sensors and internal knowledge of the system to plan and execute, based on these high-level goals, the low-level actions that are necessary to achieve these goals. The internal knowledge of the system is often an architectural model of the managed element, and the goals are usually expressed using Event Condition Action (ECA) policies, goal policies or utility function policies [40].

- **ECA policies** take the form “when event occurs and condition holds, then execute action”, e.g. when 95% of web servers response time exceeds 2s and there are available resources, then increase number of active web servers. They have been intensely studied for the management of distributed systems. A notable example is the PONDER policy language [41]. A difficulty with ECA policies is that when a number of policies are specified, conflicts between policies can arise that are hard to detect. For example, when different tiers of a multi-tier system (e.g. web and application server tiers) require an increased amount of resources, but the available resources cannot fulfill the requests of all tiers, a conflict arises. In such a case, it is unclear how the system should react, and an additional conflict resolution mechanism is necessary, e.g. giving higher priority to the web server. As a result, a considerable amount of research on conflict resolution has arisen [42, 43, 44]. However, a complication is that the conflict may only become apparent at run-time.
- **Goal policies** are more high level in that they specify criteria that characterise desirable states, but leave to the system the task of finding how to achieve that

state. For example, we could specify that the response time of the web server should be under 2s, while that of the application server under 1s. The autonomic manager uses internal rules (i.e. knowledge) to add or remove resources as necessary to achieve the desirable state. Goal policies require planning on the part of autonomic manager and are thus more resource-intensive than ECA policies. However, they still suffer from the problem that all states are classified as either desirable or undesirable. Thus when a desirable state cannot be reached, the system does not know which among the undesirable states is least bad.

- **Utility functions** solve the above problem by defining a quantitative level of desirability to each state. A utility function takes as input a number of parameters and outputs the desirability of this state. Thus, continuing our example, the utility function could take as input the response time for web and application servers and return the utility of each combination of web and application server response times. This way, when insufficient resources are available, the most desirable partition of available resources among web and application servers can be found. The major problem with utility functions is that they can be extremely hard to define, as every aspect that influences the decision by the utility function must be quantified. Research is being carried out on using utility functions, particularly in automatic resource allocation [45] or adaptation of data streaming to network conditions [46].

Monitoring

The monitoring component of the MAPE-K loop involves capturing properties of the environment (either physical or virtual, e.g. a network) that are of significance to the self-* properties of the system. The software or hardware components used to perform monitoring are called sensors. For instance, network latency and bandwidth measure the performance of web servers, while database indexing and query optimisation affect the response time of a DBMS, which can be monitored. The Autonomic Manager requires appropriate monitored data to recognise failure or sub-optimal performance of the Autonomic Element, and effect appropriate changes.

The types of monitored properties, and the sensors used, will often be application-specific, just as actuators used to execute changes to the Managed Element are also application-specific. Autonomic computing systems are based on two types of monitoring as follows.

- **Passive monitoring.** Passive monitoring systems do not require any measurement code in the system to be added, but rather observe the actual interaction of the running system. For example, passive monitoring tools exist for most operating systems, e.g. Windows 2000/XP returns memory and cpu utilisation statistics.
- **Active monitoring.** Active monitoring means engineering the software at some level, e.g. modifying and adding code to the implementation of the application or the operating system, to capture function or system calls. This can often be to some extent automated. For instance, ProbeMeister can insert probes into the compiled Java bytecode.

More recent work has examined how decide which subset of the many performance metrics collected from an dynamic environment can be obtained from the many performance tools available to it (e.g. dproc). Interestingly they observe that a small subset of metrics provided 90% of their application classification accuracy [47]. Agarwala et al. [48] propose QMON, an autonomic monitor that adapts its monitoring frequency and data volumes so to minimise the overhead of continuous monitoring while maximising the utility of the performance data. That is, an autonomic monitor for autonomic systems.

Planning

The planning aspect of the autonomic loop involves taking into account the monitoring data from the sensors to produce a series of changes to be effected on the managed element. For instance, event-condition-action (ECA) rules directly produce adaptation plans from specific event combinations. Examples of such policy languages and applications in autonomic computing include [41, 49, 50, 51, 52, 53, 54].

However, applying this approach in a stateless manner, i.e. where the autonomic manager keeps no information on state of the managed element, and relies solely on the current sensor data to decide whether to effect an adaptation plan, is very limited. Indeed, it is far better for the autonomic manager to keep information on the state of the managed element in a context model that can be updated progressively through sensor data and reasoned about.

Regarding the state information that the autonomic manager should keep about the managed element, much research has examined model-based approaches. In these approaches some form of model of the entire managed system is used by the autonomic manager. The model may also represent some aspect of the operating environment in which the managed elements are deployed, where operating environment can be understood as any observable property (by the sensors) that can impact its execution, e.g. end-user input, hardware devices, network connection properties.

The model is updated through sensor data and used to reason about the managed system to plan adaptations. A great advantage of a model-based approach to planning is that, under the assumption that the model correctly mirrors the managed system, the architectural model can be used to verify that system integrity is preserved when applying an adaptation, i.e. we can guarantee that the system will continue to operate correctly after the planned adaptation has been executed [55]. This is because changes are planned and applied to the model first, which will show the state of the system resulting from the adaptation, including any violations of constraints or requirements of the system present in the model. If the new state of the system is acceptable, the plan can then be effected onto the actual managed system, thus ensuring that the model and implementation are consistent with respect to each other.

The use of the model-based approach does not however necessarily eliminate ECA rules. Indeed, repair strategies of the architecture model may be specified as ECA rules, where an event is generated when the model is invalidated by sensor updates, and an appropriate rule specifies the actions necessary to return the model to a valid state, i.e. the adaptation plan.

In practice however, there is always a delay between the time when a change

occurs in the managed system and this change is applied to the model. Indeed, if the delay is sufficiently high and the system changes frequently, an adaptation plan may be created and sent for execution under the belief that the actual system was in a particular state, e.g. a web server overloaded, when in fact the system has already changed in the meantime and does not require this adaptation anymore (or requires a different adaptation plan) [56].

Knowledge

The knowledge in an autonomic system can come from sources as diverse as the human expert (in static policy based systems [57]) to logs that accumulated data from probes charting the day-to-day operation of a system to observe its behaviour, which is used to train predictive models [58, 59]. This section lists some of the main methods used to represent Knowledge in autonomic systems.

- **Concept of Utility.** Utility is an abstract measure of usefulness or benefit to, for example, a user. Typically a systems operation expresses its utility as a measure of things like the amount of resources available to the user (or user application programs), and the quality, reliability or accuracy of that resource etc. For example in an event processing system allocating hardware resources to users wishing to run transactions, the utility will be a function of allocated rate, allowable latency and number of consumers, e.g. [11]. Another example is in a resource provisioning system where the utility is derived from the cost of redistribution of workloads once allocated or the power consumption as a portion of operating cost [60, 61].
- **Reinforcement learning.** Reinforcement learning is used to establish policies obtained from observing management actions. At its most basic it learns policies by trying actions in various system states and reviewing the consequences of each action [62]. The advantage of reinforcement learning is that it does not require an explicit model of the system being managed, hence its use in autonomic computing [63, 64]. However it suffers from poor scalability in trying to represent large state spaces, which also impacts on its time

to train. To this end, a number of hybrid models have been proposed which either speed up training or introduce domain knowledge to reduce the state space, e.g. [65, 66].

- **Bayesian Techniques.** As well as rule-based classification of policies to drive autonomy, probabilistic techniques have been used throughout the self-management literature to provide a way to select from numbers of services or algorithms etc. For example, Guo [67] shows how Bayesian Networks (BNs) are used in autonomic algorithm selection to find the best algorithm, whereas cost sensitive classification and feedback has been used to attribute costs to self-healing equations to remedy failures [68].

Using knowledge about the system configuration, a problem- diagnosis component (for example, based on a Bayesian network) would analyze information from log files, possibly supplemented with data from additional monitors that it has requested. The system would then match the diagnosis against known software patches (or alert a human programmer if there are none), install the appropriate patch, and retest.

All the techniques presented in this and the above sections contribute to increasingly achieve sophisticated autonomic managers for managed elements. Ultimately, the distinction between the autonomic manager and the managed element may become merely conceptual rather than architectural, or it may melt away, leaving fully integrated, autonomic elements with well-defined behaviors and interfaces, but also with few constraints on their internal structure.

2.3 Model Driven Development

Model Driven Development (MDD) is a paradigm where models are central in the development. Model Driven Architecture (MDA) is a framework for software development proposed by the Object Management Group (OMG) in 2001 [69] (i.e., MDA is a concrete realization of MDD). The notion of Model Driven Engineering (MDE) emerged later as a paradigm generalizing the MDA approach for software development [15].

2.3.1 Definition

The arrival of the MDD and MDA are changing the way of using models in the development of software. Model-driven is a paradigm where models are used to develop software. This process is driven by model specifications and by transformations among models. It is the ability to transform among different model representations that differentiates the use of models for sketching out a design from a more extensive model-driven software engineering process where models yield implementation artifacts. As stated by Agrawal [70]:

“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 approaches is to automatically translate an abstract specification of the system into a fully functional software product.

2.3.2 Model Driven Software Development Initiatives

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 [71]. Models have been used for a long time in the software development field. From formal and executable specification languages (like OBLOG [72], TROLL [73] or OASIS [74]), to the most accepted notations (like UML [75]) and processes (like RUP [76]) models are present in the software development area.

Stuart Kent [15] defines Model Driven Engineering (MDE) by extending MDA with the notion of software development process (that is, MDE emerged later as a generalization of the MDA for software development). MDE refers to the systematic

use of models as primary engineering artifacts throughout the engineering lifecycle. Kurtev provides a discussion on existing MDE processes [77] (refer to [78, 79] for a specific approach). In general, these approaches introduce concepts, methods and tools [80]. 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. MDA classifies models into two classes: Platform Independent Models (PIMs) and Platform Specific Models (PSMs) [81]. 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 [81]. Doing so, the definition of platform becomes fundamental.

Although the contribution of MDA has been critical, other initiatives under different descriptive terms have pushed on the MDSD direction. These initiatives (or specic paradigms) highlight distinct aspects and/or follow specic strategies for applying MDSD. The following are remarkable examples of these initiatives.

- **Automatic programming.** According to Balzer [82], who is considered the initiator of the modern automatic programming paradigm, automatic programming is based on the use of methods and tools which support the acquisition of high level of abstraction specifications, their validation and the generation of executable code. He was focused on the generation of efficient implementations, since the hardware resources (CPU power, memory size, etc.) were limited. Therefore, he proposes a semi-automated (interactive) translation approach which facilitates the specification of optimizations by human developers. It is important to note that he considers that the application of this paradigm to a narrower area (like expert systems) allows an “attempt to eliminate the need for interactive translations”.
- **Generative Programming.** This paradigm was proposed by Czarnecki in his PhD Thesis [83] although the term was coined by Eisenecker in [84]. In Eisenecker words, Generative Programming “is a comprehensive software development paradigm to achieving high intentionality, reusability, and adapt-

ability without the need to compromise the run-time performance and computing resources of the produced software”. It is highly based on domain specific engineering and product line development, using techniques like generic programming, domain-specific languages and aspect-oriented programming. Unlike other more general paradigms, Generative Programming suggests very specific techniques and steps for developing methods which follow this approach.

In general, MDSD initiatives promote a paradigm of reuse and automation. This emerges through the extensive use of models and model transformations, which replaces cumbersome (and usually repetitive) implementation activities. In this way, model-driven approaches improve development practices by accelerating them.

2.3.3 Domain Specific Languages

Domain specific languages play a key role in several of the MDSD approaches that have been presented above. According to [85], a domain specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

DSLs are not a new topic, but the current stress on MDSD have focused the interest of both academy and industry on this kind of languages. Examples of DSLs abound, including well-known and widely-used languages such as LATEX, YACC, Make, SQL, and HTML. As state by [85], the older programming languages (Cobol, Fortran, Lisp) all came into existence as dedicated languages for solving problems in a certain area (respectively business processing, numeric computation and symbolic processing).

DSLs are tightly related to the Domain Engineering. In words of Tolvanen [86], the main focus of Domain Engineering is finding and extracting domain terminology, architecture and components. It is important to note that two points of view when dealing with the domain concept can be considered, as highlighted by Simos [87].

- **Conceptual domain.** From this point of view, a domain is a set of inter-related real-world concepts. For instance, the health-care domain contains

concepts like medical center, patient, disease, medicament, etc. As another example, the industrial factory domain contains concepts like stock, supplier, client, worker, etc.

- **Systems domain.** From this point of view, a domain is characterized by a set of systems that share some common features [87]. These systems usually address a common problem area and conceivably share a common solution structure. In this case, we can talk about the expert systems domain, the database-based systems domain, the control/monitoring systems domain, the software games domain, etc.

Note that a software system can be seen as the combination of both a conceptual domain and a system domain. For instance, we can find experts system for health-care and control/monitoring systems for industrial factories, but also exists expert systems for industrial factories and control/monitoring systems for health-care. Specific languages exists both for conceptual domains and systems domains.

Many benefits due to the use of DSLs can be found in the literature. For instance, according to [85].

- DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs.
- DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes.
- DSLs enhance productivity, reliability, maintainability, and portability.
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge.
- DSLs allow validation and optimization at the domain level.

But some drawbacks have been also identified. These drawbacks are related to the associated costs (for designing, implementing and learning the DSL) and

the specific nature of the language (possible lack of expressiveness and/or loss of efficiency).

Some researchers suggest that the success of visual notations as commonly used domain-specific languages is contingent on making similar tools and concepts for visual languages a commodity that can be readily used and understood by a wide audience, effectively lowering the initial hurdle to adoption [88]. Hopefully, the number and quality of tools for implementing DSLs is growing and, therefore, a widely use of DSLs could be foreseen.

2.4 Software Product Lines

Mass production was popularized by Henry Ford in the early 20th Century. McIlroy coined the term software mass production in 1968 [89]. It was the beginning of Software Product Lines. In 1976, Parnas introduced the notion of software program families as a result of mass production [90]. The use of features (to drive mass production) was proposed by Kang in the early 1990s [91]. Shortly, the first conferences appeared turning SPL into a new body of research [92, 93].

2.4.1 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” [16]. This definition can be redefined into five major issues:

1. **Products.** 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.** Features are units (i.e., increments in application functionality) by which different products can be distinguished and defined within an SPL [94].
3. **Domain.** 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 [95].
4. **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 [16].
5. **Production Plan.** 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 [96]. The production plan ties together all the reusable assets to assemble (and build) end products. Synthesis is a part of the production plan.

2.4.2 Software Product Line Processes

Software product lines (or system families) provide a highly successful approach to strategic reuse of assets within an organization. A standard software product line consists of a product line architecture, a set of software components and a set of products. A product consists of a product architecture, derived from the product line architecture, a set of selected and configured product line components and product specific code.

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 processes: domain engineering (also called core asset development), application engineering (also called product development) and management. These three processes are complementary and provide feedback to each other.

- **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” [97]. That is, Domain engineering is, among others, concerned with identifying the commonality and variability for the products in the product line and implementing the shared artefacts such that the commonality can be exploited while preserving the required variability.

Using a “design-for-reuse” approach, domain engineering (core asset development [16]) is on 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.

Application Engineering is “the process of building a particular system in the domain” [97]. Application engineering (a.k.a., product Development [16]) is responsible for deriving a concrete product from the SPL using a “design-with reuse” approach. To achieve this, it reuses the reusable assets developed previously.

During application engineering, individual products are developed by selecting and configuring shared artefacts and, where necessary, adding product-specific extensions. This process is subdivided into application analysis, application design and application implementation.

Management is a separated process where organizational issues are handled specifically [16]. This process is responsible for giving resources, coordinating, and supervising domain and application engineering activities.

See [16, 98] for more details about the above processes. In SPL processes, variability is made explicit through variation points. A variation point represents a delayed design decision. When the architect or designer decides to delay the design decision, he or she has to design a variation point. The design of the variation point requires several steps: (1) the separation of the stable and variant behaviour, (2) the definition of an interface between these types of behaviour, (3) the design of a variant management mechanism and (4) the implementation of one or more variants. Given a variation point, it can be bound to a particular variant. For each variation point, the set of variants may be open, i.e. more variants can be added,

or closed, i.e. no more variants can be added. Overall, during domain engineering new variation points are introduced, whereas during application engineering these variation points are bound to selected variants

Behind the software product line approach we can find the economies of scope principle. While economies of scale arise when multiple identical instances of a single design are produced collectively, economies of scope arise when multiple similar but distinct designs are produced collectively [99]. In this context, the same practices, processes, tools and materials are used to design and build similar unique products. This methodical reuse is the responsible productivity and quality increase.

2.4.3 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 run-time 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 run-time 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.

In modern computing and network environments, a high degree of adaptability from software systems is demanded. Computing environments, user requirements and interface mechanisms between software and hardware devices like sensors may change dynamically during run-time. Therefore, in these kinds of dynamic environments, application of SPL needs to be changed from a static perspective to a dynamic perspective, where systems capable of modifying their own behavior with respect to changes in its operating environment are achieved by dynamically re-binding variation points at run-time. This is the idea of Dynamic Software Product Lines (DSPL) [18].

DSPL development mainly intends to produce configurable products [100] 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 run-time binding. A CP may be considered as an extension to traditional SPL products where there are no bound features but the decision maker and the reconfigurator and the remaining features are bound at run-time. As a consequence, new features may be added to an existing product or even existing features may be updated at run-time.

Interest in DSPLs is growing as more developers apply the SPL approach to dynamic systems. The first workshop on DSPLs was held at the 11th International Software Product Line Conference in Kyoto in 2007, and currently, the workshop on DSPLs is in its fourth edition.

2.5 Conclusions

The purpose of this chapter was to provide a brief introduction to the existing background on top of which this work is built on. Inspired by biology, **Autonomic Computing** has evolved as a discipline to create software systems and applications that self-manage in a bid to overcome the complexities and inability to maintain current and emerging systems effectively. **Model Driven Development** is a paradigm to develop programs based on modelling. Software models are specified, from which other models or even code are derived. This paradigm eases cumbersome and repetitive tasks, and achieves productivity gains. **Software Product Lines** offer a paradigm to develop a family of software products. The focus shifts from the development of an individual program to the development of reusable assets that are used to develop a family of programs.

TLAs You Need

OMG: The Object Management Group is an international, not-for-profit industrial consortium that creates and maintains software interoperability specifications.

UML: The Unified Modelling Language is an industry standard visual language for modelling software systems. These models capture knowledge about a system at various abstraction levels, ranging from requirements and analysis models to design models.

MDA: The Model-Driven Architecture is a set of OMG standards that enables the specification of models and their transformation into other models and complete systems.

MDD: Model Driven Development is an emerging paradigm for software construction that uses models to specify programs, and model transformations to synthesize executables.

DSL: A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

OWL: The Web Ontology Language is an ontology markup language that enables context sharing and context reasoning. In the artificial intelligence literature, an ontology is a formal, explicit description of concepts in a particular domain of discourse.

XMI: The XML Metadata Interchange is an OMG standard for exchanging metadata information via Extensible Markup Language (XML). The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages (metamodels).

SPL: A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

CVL: The common language of variability expresses variability in a language independently of the base modelling language. This base-model can be a domain-specific language as well as a general purpose languages like UML.

TLA: Three-letter acronym.

Chapter 3. STATE OF THE ART

“If I have seen farther than others, it is because I was standing on the shoulder of giants.”

– Isaac Newton (1643-1727).

3.1 Overview of the Chapter

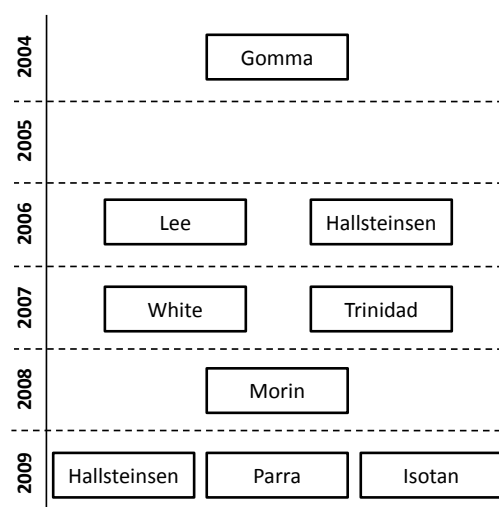


Figure 3.1: Scope of Chapter 3

Dynamic system reconfiguration refers to making changes to a deployed system after it has started operation. Dynamic addition, deletion, or modification of system features, or dynamic changes of architectural structures [101] are some examples of dynamic reconfiguration. This dynamic system reconfiguration has been studied in various research areas such as self-healing systems [102, 103, 55], context-aware computing [4, 5] or ubiquitous computing [6, 7]. When a change in the operational context is detected,

it may trigger system reconfiguration to accommodate context events or to meet quality requirements.

However, dynamic reconfiguration approaches in the literature have focused on reconfiguration of a single system, not on a family of systems. That is, accommodation of system-specific dynamic needs that may differ from one system to another.

Different from statically configured systems, a reconfigurable system family is able to: (1) monitor the system operational context, (2) validate a reconfiguration

request with consideration of change impacts and available resources, (3) determine strategies to handle currently active services during reconfiguration, and (4) perform dynamic reconfiguration while maintaining system integrity.

In this chapter, we present an analysis of the most important approaches that have been proposed to support run-time reconfiguration of system families. These run-time reconfigurations enable the system to adhere to different degrees of Autonomic Computing. To analyse these approaches, we suggest criteria to evaluate both the achieved autonomic behaviour and the methodology to achieve this behaviour. Finally, we discuss the resulting analysis and we also identify patterns in the reconfiguration infrastructures.

3.2 Classification Criteria

This section provides criteria to classify system family approaches that achieve some sort of autonomic behavior. Specifically, the three first criteria evaluate the achieved autonomic behaviour, and the last criterion evaluates the methodology to achieve the autonomic behaviour.

3.2.1 Adoption Level of Autonomic Computing

This criterion is based on the scale proposed by IBM to evaluate the adoption of autonomic computing [2]. IBM has proposed a set of Autonomic Computing Adoption Levels that spans from Level 1: Basic, to Level 5: Autonomic. Briefly, these levels are presented as follows.

- **Level 1** defines the state whereby system elements are managed by highly skilled staff who utilise monitoring tools and then make the require changes manually. IBM believes this is where most IT systems are today.
- **Level 2** is known as Managed. This is where the system's monitoring tools collage information in an intelligent enough way to reduce the systems administration burden.

- **Level 3** is entitled Predictive whereby more intelligent monitoring than Level 2 is carried out to recognise system behaviour patterns and suggest actions approved and carried out by IT staff.
- **Level 4** is the adaptive level. Here the system uses the types of tools available to Level 3 system's staff but is more able to take action. Human interaction is minimised and it is expected that the performance is tweaked to meet service level agreements.
- **Level 5** is the full autonomic level, where systems and components are dynamically managed by business rules and policies, thus freeing up IT staff to focus on maintaining ever changing business needs.

Since we focus on self-managing systems we are precluding work that would conform to Levels 1 through to 3 and focus on what would be deemed by IBM as Adaptive and Autonomic Computing only (Levels 4 and 5).

3.2.2 Relevance of the Autonomic Computing

This criterion evaluates the relevance of the autonomic behavior in comparison with the overall functionality of the system. Autonomic computing can play the role of the *core* functionality or it can play the role of *supporting* functionality. Both Core and Support relevance are described as follows.

- **Core.** This is where the self-management function is driving the core application itself. That is, if the application's focus is to deliver multimedia data over a network and the work describes an end-to-end solution including network management and display, audio, etc., then we identify the self-management as core.
- **Support.** This is where the self-management function focuses on one particular aspect or component of the architecture to help improve the behaviour of the complete architecture using autonomicity. For example, focus on resource management or network support only.

3.2.3 Reinforcement of the Autonomic Knowledge

This criterion evaluates whether the knowledge that drives the autonomic behaviour is static (Autonomous reinforcement) or it is updated with context information (Autonomic reinforcement). On the one hand, Autonomous reinforcement provides the same system response to a particular event always. On the other hand, Autonomic reinforcement can provide different responses to a particular event depending on the current state of the system knowledge.

- **Autonomous.** This is where the system self-adapts to the environment to overcome challenges that require adaptation, but it is not feeding its own knowledge with context information in order to better fit next adaptations.
- **Autonomic.** This is where not only higher-level human based policies are taken into account, but the knowledge is feeded with context information in order to adapt itself accordingly.

Finally, there is also possible that the system would evolve the policies that drives the system depending on how well the “old” policies did. This is connected to the work carried out in Artificial Intelligence, which is an area of research that falls out of the scope of this work.

3.2.4 Maturity of the 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. The *maturity* criterion [104] presents four typical stages of software engineering for dynamic adaptation as follows.

- **Stage 0:** non-adaptive systems. In this stage, the system realizes no kind of dynamic adaptation. This applies only to those systems that do not (need to) adapt to any kind of environmental changes.
- **Stage 1:** implicit adaptation. Most systems are at least at evolution this stage one. At this stage, the adaptation behavior is modeled as indistinguishable

part of the functionality. Any system at this evolution stage or beyond can be considered an adaptive system. The motivation to use dynamic adaptation at this stage is mainly the necessity to adapt to dynamic environments. If we regard a vehicle stability controller, it is necessary to estimate the current driving situation. Decisions and control strategies then depend on this context information. This example is implicit dynamic adaptation, since there is definitely an adaptation although most developers do neither know that they currently develop an adaptive system nor that they have an idea of the implications of dynamic adaptation. Since the adaptation behavior is not explicitly modeled, adaptations often happen locally at a component level. The dependencies between different components cannot be captured and are often not considered at all. This leads to serious problems since adaptations in one component usually have an influence on the quality of the provided services of the component. Not communicating this influence to relying components often leads to serious failures. The latter are difficult to reconstruct and it is hardly possible to identify the causing faults.

- **Stage 2:** explicit adaptation, no engineering of adaptation. Starting at stage 2, dynamic adaptation is explicitly considered in system development. Most of the research of recent years has been focused on this stage. Also in industry some systems have already reached this stage. The main characteristic that makes a system belonging to this stage is the presence of a dedicated run-time adaptation framework. This framework could be a central component in the system coordinating all adaptation 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. For industry, the main reason to evolve into this stage is the system quality. Some companies already noticed that implicitly used dynamic adaptation is a major cause for the troubles they have. 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. The complexity of dynamic adaptation that has been neglected at stage 1 is now made visible. Although the quality problem can be encountered, an immense effort is required to manage the complexity of the adaptation behavior.

- **Stage 3:** software engineering of adaptive systems. This constitutes the currently final stage. In this stage not only an execution platform or mechanism to realize dynamic adaptation at run-time is provided, but also a dedicated methodology enabling developers to systematically develop adaptive embedded systems. First, this includes a seamless modelling methodology. In this regard, it is important to make the complexity manageable, e.g. by supporting the modular and hierarchical definition of adaptation. Second, the seamless software engineering approach also includes the model based analysis, validation and verification of dynamic adaptation. For dependable 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. Furthermore, this stage also benefits from the whole range of typical gains brought by model-driven engineering (MDE) approaches (i.e. validation, verification, reuse, automation). 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.

The above stages enable us to evaluate to what extent an approach provides a methodology to guide the developer systematically from the requirements to a validated and verified adaptive system.

3.3 Analysis of Approaches for System Family Reconfiguration

In this section, we use the above criteria to analyse the most relevant approaches for System Family Reconfiguration, paying special attention to how the variability is managed. The approaches are presented chronologically according to the year in which they appeared. For each one of these approaches, we present the following information:

- A description of the Variability Specification that the approach uses to describe the system family.
- The reconfiguration infrastructure provided by the approach.
- Successful case studies that the approach has carried out.

For each approach, the most relevant information is presented following the layout of Table 3.1. Top of Table 3.1 shows the *Scope*, *Variability Specification* and *Reconfiguration Infrastructure* of the approach. Bottom of Table 3.1 shows the classification of the approach according to the criteria introduced on Section 3.2.

Approach Authors - Approach Name			
Scope	Scope of the approach.		
Variability Specification	Approach techniques for Variability Specification.		
Reconfiguration Infrastructure	Approach infrastructure for reconfiguration.		
AC Adoption	[Level 4 Level 5]	AC Relevance	[Core Support]
Reinforcement	[Autonomous Autonomic]	Maturity	[Stage 0 Stage 1 Stage 2 Stage 3]

Table 3.1: Template for Approach Classification.

3.3.1 Gomaa and Hussein Approach

Gomaa and Hussein **Gomaa and Hussein** (see Table 3.2) address the problem of dynamic system reconfiguration by changing the configuration of the running system from one member of the product family to another. Specifically, they focus on changing the application configuration at run-time after it has been deployed from the software product line.

In order to support dynamic software reconfiguration, the Reconfigurable Evolutionary Product Family Life Cycle (REPFLC) is a new life cycle which builds on previous research into software product families [105] and extends it significantly to support dynamic reconfiguration. Figure 3.2 right shows the REPFLC.

The REPFLC method consists of three major activities: (1) Product Family Engineering. (2) Target System Configuration. (3) Target System Reconfiguration.

1. During **Product Family Engineering**, similarities and variations among the members of the product family are established through modelling and analysis of the product family requirements. By considering appropriate software patterns in the product family, members of the product family are designed to be reconfigurable using the configuration change management modelling method. The product family architecture is designed in terms of components and their interconnections (see Figure 3.2 left).
2. During the first **System Configuration**, the components of the product family are configured on the basis of user-required features.
3. During **System Reconfiguration**, users can specify run-time configuration changes so that an executable system is dynamically changed from the old configuration to the new configuration.

To support reconfiguration, Gomaa and Hussein suggest to design components in order to be capable of transitioning to a state where it can be reconfigured. In reconfigurations, these components can be manipulated by means of *Reconfiguration Commands*. Reconfiguration commands describe reconfiguration actions associated

with user required changes, or reconfiguration scenarios. The reconfiguration commands are passivate, checkpoint, unlink, remove, create, link, activate, restore, and reactivate.

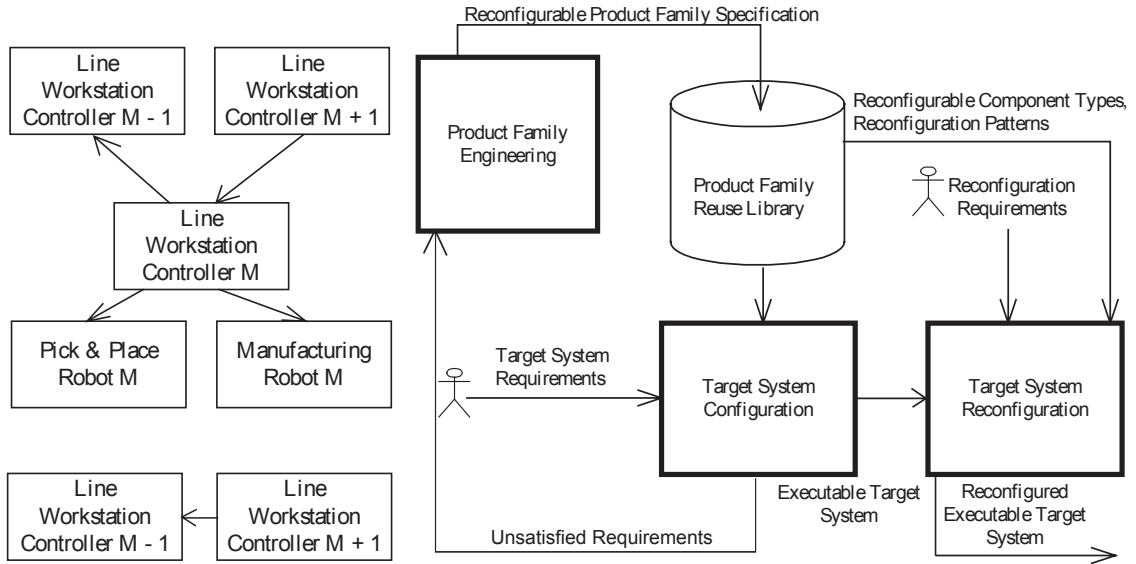


Figure 3.2: Gomaa and Hussein Approach

For example, a component that receives a passivate command with no parameters must eventually transition to a passive state. If the command has parameters, the component must go idle, i.e., be inactive as long as there is an interconnection with the components denoted by the parameters.

To support this reconfiguration approach a proof-of-concept prototype has been developed: the Reconfigurable Product Line UML Based Software Engineering Environment (RPLUSEE). The RPLUSEE prototype uses the commercial Rational Rose Real Time¹ (Rose RT). Components are mapped to Rose RT capsules. In Rose RT, capsules execute Rose RT statecharts which represent transitions as events guarded by conditions. Actions are implemented with Rose RT functions and C++ code. Capsules communicate through exchange of messages sent and received through ports.

Two product families were developed using the REPFLC method and the RPLUSEE tool in order to validate the approach.

¹<http://www-01.ibm.com/software/awdtools/developer/technical/>

- A reconfigurable automobile cruise control product family was designed.
- A reconfigurable factory automation product family architecture was designed and implemented.

As part of the validation process, three techniques, provided by the model execution capability of Rose RT, were employed: execution control, visual component instance monitoring, and analysis of message trace outputs. The validation process confirmed that reconfiguration scenario change transactions executed correctly and that component reconfigurations took place as planned.

Gomaa and Hussein - REPFLC Approach			
Scope	System automation: cruise and factory control		
Variability Specification	Architecture model designed in terms of components and their interconnections.		
Reconfiguration Infrastructure	Reconfiguration Commands implemented by Rational Rose RT functions and C++ code.		
AC Adoption	Level 4	AC Relevance	Support
Reinforcement	Autonomous	Maturity	Stage 2

Table 3.2: Classification of Gomaa and Hussein Approach.

3.3.2 Lee and Kang Approach

Lee and Kang [106] (see Table 3.3) introduce a *feature binding analysis* step in SPLs to achieve the development of dynamically reconfigurable core assets. *Feature binding analysis* consists of two activities: feature binding unit identification and feature binding time determination. These activities refine feature models through grouping of features into feature binding units that has the same binding time.

Once features are grouped into feature binding units, their binding times are determined. In Lee and Kang approach, feature binding time is analyzed based on two view points: the product lifecycle view, in which the focus is given to the lifecycle phase in which a feature is incorporated into a product, and the binding state view, in which the focus is given to represent the inclusion, availability, and

activation states of features. (Note that a feature may not be available for use even if it is physically included in a product.)

The refined feature model from the *feature binding analysis* (see Figure 3.3 left) is the key design driver to develop the product line reconfigurable components. For dynamic reconfiguration of feature binding units, variation points corresponding to each binding unit should be identified in the design component model and implemented with appropriate binding techniques. For example, dynamic binding of objects, menus, and plugins are techniques that support dynamic binding their reconfigurable components.

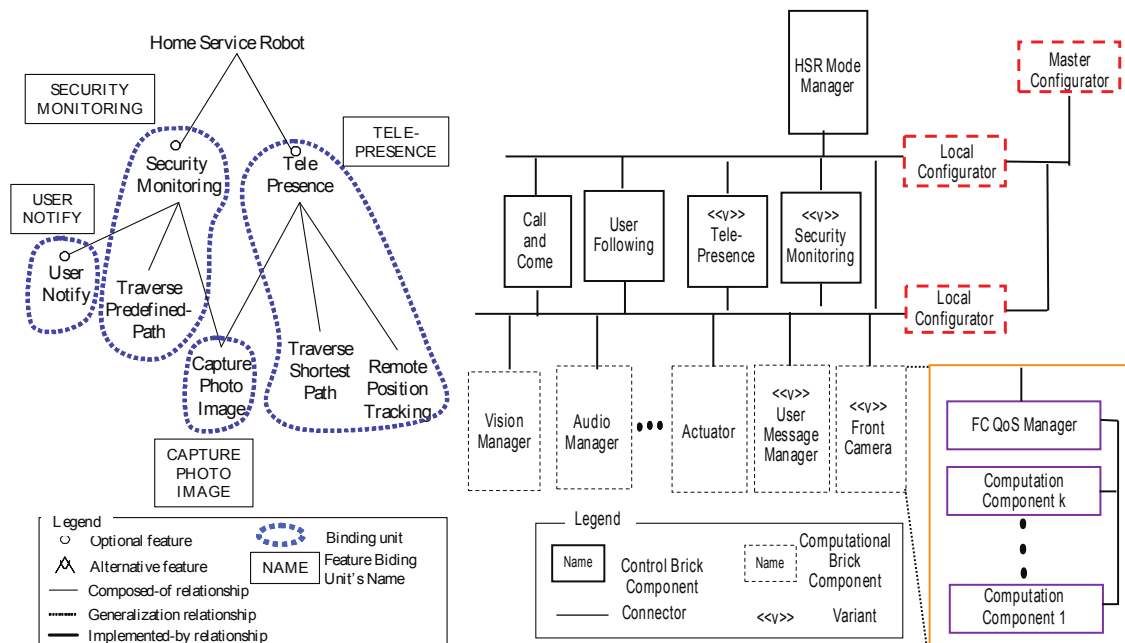


Figure 3.3: Lee and Kang Approach

As a result of the above activities, a set of reconfigurable components is obtained. These components are reconfigured following a *dynamic reconfiguration strategy* in response to context events. That is, decisions of when to start a reconfiguration are analyzed through an operational context analysis. This context analysis consists of three subactivities: contextual parameter identification, situation definition, and mapping of each situation to a reconfiguration request.

Given a reconfiguration request, the *dynamic reconfiguration strategy* is about “how” to perform dynamic reconfiguration. The strategy is specified with consider-

ation of binding dependencies (i.e., require and exclude), change impacts to other binding units, and required resources (e.g., components). The reconfiguration strategy is specified for six reconfiguration phases as follows: (1) check pre-conditions, (2) send a Suspend event to currently active binding units that are involved in reconfiguration, (3) remove or parameterize binding units that have to be deleted or changed, (4) instantiate and bind binding units that are newly added, (5) check post-conditions, and (6) resume suspended binding units and start newly added binding units.

The execution of the *dynamic reconfiguration strategy* depends on a Master Configurator and Local Configurators (see Figure 3.3 right). The Master Configurator is responsible for monitoring the context and product status, and processing reconfiguration requests. The Local Configurators provides Master Configurator with product state information by analyzing messages at each connector and executes reconfiguration commands received from the Master Configurator. There is a prototype implementation of the *dynamic reconfiguration strategy* using this Master Configurator and Local Configurators approach [107].

Lee and Kang overall approach has been applied to the development of home service robot control software. These home service robots (HSR) utilize various technology-intensive computational components such as speech recognizers, vision processors, and actuators to offer feature binding units.

Lee and Kang - Feature Binding Units			
Scope	Pervasive systems: home service robot control software		
Variability Specification	Feature models refined into feature binding units that has the same binding time.		
Reconfiguration Infrastructure	Reconfigurable components and dynamic reconfiguration strategy (supported by Master Configurator and Local Configurators)		
AC Adoption	Level 5	AC Relevance	Support
Reinforcement	Autonomic	Maturity	Stage 2

Table 3.3: Classification of Lee and Kang Approach.

3.3.3 Hallsteinsen et al. (MADAM) Approach

Hallsteinsen et al. [108] (see Table 3.4) present the MADAM approach to building adaptive systems. They target distributed applications accessed through handheld networked devices which have to adapt to context changes such as variation in network capacity and periods of network absence, hands and eyes becoming temporarily busy with other things, batteries running low, and devices running out of memory.

The MADAM approach is based on ideas from software product line engineering. Adaptive applications are built as component oriented system families with variability modeled explicitly as part of the family architecture. By representing the family architecture at run-time, they are able to offload much of the complexity of adaptation to a generally reusable adaptation platform.

Hallsteinsen et al. extend SPLs by adding the ability to automatically derive changed configurations by monitoring the context, and to automatically reconfigure the application while it is running.

The adaptation platform of the MADAM approach provides (1) a *conceptual model* and (2) *reference architecture* for adaptive applications as follows.

1. The *conceptual model* (see Figure 3.4 left) is based on entities which interact with other entities by providing and making use of services through ports. A port represents a service offered by an entity or a service needed by an entity. Entities may be composed of smaller entities, allowing for a hierarchic structure. To model variation, both in the application and in its context, the *conceptual model* provides the concept of entity type. An entity type defines a class of entities with equivalent ports which may replace each other in a system. With these concepts the *conceptual model* is able to model an adaptive application architecture as a possibly hierarchic composition of entity types, which defines a class of application variants as well as a class of contexts in which they may operate.
2. The *reference architecture* (see Figure 3.4 right) provides components for monitoring user needs and available resources, for deriving a more suitable variant when the user needs or available resources change such that the current vari-

ants is rendered unsuitable, and for transforming the current variant into the preferred one by reconfiguration at the component level. To enable the derivation of the variant that best fits a given context, the MADAM approach is based on property annotations associated with ports. Property annotations allow us to reason about how well an application variant matches its context, by comparing the properties of the services provided by the application with the properties required by the user and the properties expressing the resource needs of the application with the property annotation describing the resources provided by the current computing infrastructure. The match to user needs is expressed in a utility function. By default the utility function is a weighted mean of the differences between properties representing user needs and properties describing the service provided by the application, where the weights represent priorities of the user.

The conceptual model and the variability modeled there is represented by platform components. Following the reference architecture, the platform components launch adaptive applications on request (by the user or another application) and manage the running applications which are competing for the resources of the device. Specifically, the platform components monitor the context and when significant changes occur they reconfigure the running applications accordingly.

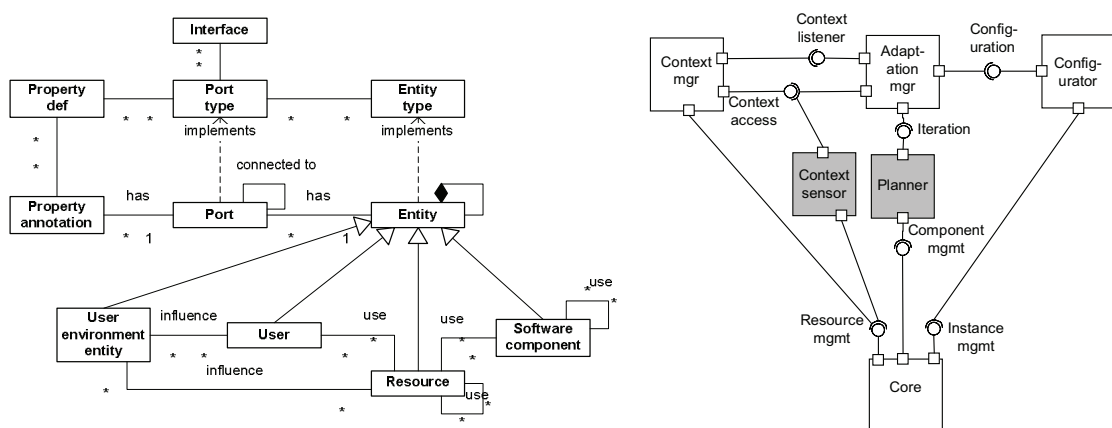


Figure 3.4: Hallsteinsen et al. Approach

To demonstrate the practical applicability and usefulness of the MADAM approach, Hallsteinsen et al. have implemented a prototype adaptation platform and

two industrial pilot applications in collaboration with the MADAM industrial partners Condat and Integrasys. The pilot applications are based on existing commercial mobile applications. The implementation was done in Java J2ME/CDC and some experiments have been done on an iPAQ 5550 in a simulated context environment.

Hallsteinsen et al. Approach - The MADAM Approach			
Scope	Pervasive systems: Mobile devices		
Variability Specification	Conceptual model based on the notion of entities connected through ports and the concept of entity type to represent variability.		
Reconfiguration Infrastructure	Property annotations associated with ports and a utility function to determine the properties which matches a given context.		
AC Adoption	Level 4	AC Relevance	Support
Reinforcement	Autonomos	Maturity	Stage 2

Table 3.4: Classification of Hallsteinsen et al. Approach.

3.3.4 White et al. Approach

White et al. [109] (see Table 3.5) address SPL that allow mobile devices to download software configurations on-demand. When a device enters a particular context, the application provider service must deduce and create a variant for the device. Given the large array of device types and rapid development speed of new devices and capabilities, the SPL will not be able to know about all device types a priori. As devices enter a context, their unique capabilities must be discovered and dealt with efficiently and correctly.

To address these SPL for online mobile software variant selection, White et al. have developed a tool called Scatter that first captures the resources of a mobile device and then constructs a custom variant from the SPL to the device. That is, they are addressing a cycle of device discovery, variant selection based on requirements, and variant deployment.

First of all, White et al. specify the variant composition rules by means of a

domain-specific language (DSL) named Scatter (see Figure 3.5 Left). Scatter allows developers to visually model (1) the components of their PLA, (2) the dependencies and composition rules of components, and (3) the nonfunctional requirements of each component.

1. The *Component* element is the basic building block in the Scatter DSL that represents an indivisible unit of functionality, such as a Java class or specific feature.
2. *Dependencies* between components can be created by specifying a composition predicate (Required, Exclusive OR, Cardinality, or Exclusion) and the Components to which the predicate should be applied.
3. The child *requirement* elements of a component specify the non-functional requirements that must be satisfied by a device's resources. Each requirement has a Name, Type, and Value attribute associated with it.

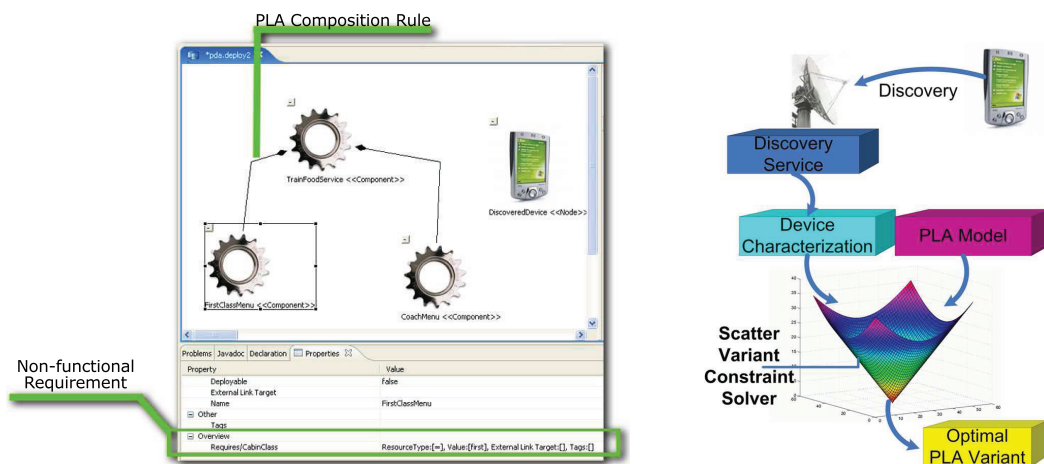


Figure 3.5: White et al. Approach

Given a Scatter specification, a compiler converts the graphical models from the Scatter modelling tool into a both a Prolog knowledge base and a Constraint Satisfaction Problem (CSP) [110] that can be operated on using a Prolog constraint solver.

This knowledge base about a system family of devices, is used by the variant selection engine to select an optional variant for a discovered mobile device. The

properties of the discovered device can be obtained from a mobile device discovery service.

The device discovery service communicates discovered devices to Scatter’s variant selection engine. The remoting mechanism allows the discovery service to report back key device nonfunctional properties, such as OS, memory, and CPU speed.

Scatter exposes a SOAP-based web service for remotely communicating device characterizations as they are discovered. The properties of a device are reported back to Scatter as key/value pairs (see Figure 3.5 right). The keys match the names of the non-functional properties constrained by the non-functional requirements in the Scatter graphical model. Then, these constraints and key/value pairs are used by the variant selection engine to filter the list of variants that can be deployed to a device.

The variant selection engine, based on a Prolog constraint solver, automatically select a correct and optimal variant for the discovered device. The Scatter selection engine feeds the device specification, provided by a discovery service, and Prolog knowledge base created by the Scatter compiler, to the constraint solver. The selection engine then translates the results from the constraint solving back into configuration decisions for the variant.

The configuration decisions determine the software components that conform the variant for the discovered mobile device. Finally, these components are send and deployed in the mobile device.

White et al. Approach - The Scatter Tool			
Scope	Pervasive systems: Mobile devices		
Variability Specification	Scatter DSL: Specification of the Components and Resources of the Mobile Device		
Reconfiguration Infrastructure	Device discovering through SOAP-based web service and variant selection by Prolog constraint solver		
AC Adoption	Level 5	AC Relevance	Core
Reinforcement	Autonomic	Maturity	Stage 2

Table 3.5: Classification of White et al. Approach.

3.3.5 Trinidad et al. Approach

Trinidad et al. [111] (see Table 3.6) argue that a SPL may be the approach that fits better to build dynamically adaptable products, because modelling techniques that represent an SPL can be used to describe all the products which can derive from the original one.

In particular, they suggest to use feature models to model the potential states or configurations of a product. Then, they propose a process for the generation of a component architecture from a feature model. The generated architecture is able to activate or deactivate features making use of a configurator component that performs some analysis operations on feature models to make decisions.

Overall, Trinidad et al. map each feature of the feature model (see Figure 3.6 left) into a component of the architecture (see Figure 3.6 right). This mapping is a two step proces: (1) Defining the core architecture (features that are common to every product), and (2) Defining the dynamic architecture (features that are specific of a particular set of products).

1. **Defining the core architecture.** To perform this mapping, they create a component for each feature. The components will connect among them depending on the relationships among features in the feature model. For each hierarchical relationship between a parent feature and a child feature, a dependency from the parent component to the child component is created. For each cross-tree constraint (depends and excludes relationships) a dependency in the direction of the constraint is created between the respective components.
2. **Defining the dynamic architecture.** To introduce dynamic adaptation in the architecture, a feature component will provide a set of interfaces that will vary from its responsibilities, and will require some functionalities to its child features by means of input interfaces. To connect the features each other, a *relationship component* will be created for each relationship in the feature model. A *relationship component* provides and requires the interfaces of the feature components that it joins, acting as an intermediary among feature components.

Furthermore, each *feature component* is coupled with *relationship components* that must be aware of any de/activation that affects the features it links. For this reason, all the relationship components must also provide a Relationship interface for the coupled features to communicate any change in their state.

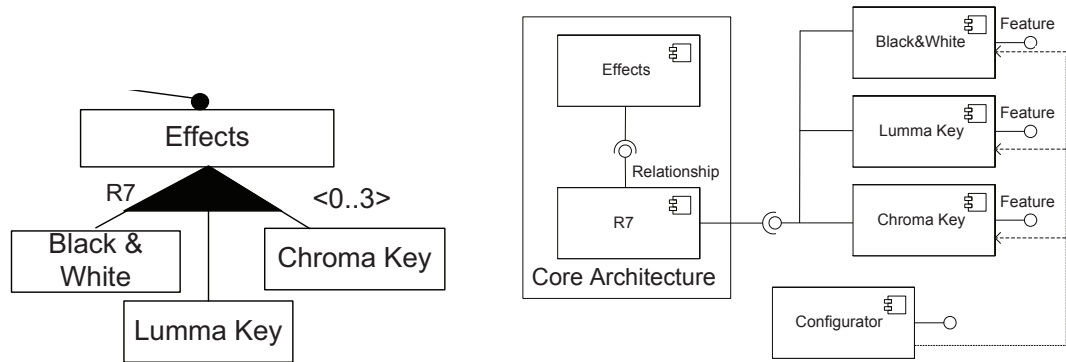


Figure 3.6: Trinidad et al. Approach

To provide the dynamic behaviour, Trinidad et al. incorporate a *reconfigurator component* in the architecture. This *reconfigurator* communicates with every feature for de/activation. This is the reason why every feature component must provide a Feature interface that allows the feature de/activation.

Determining if a target configuration is valid is also responsibility of the *reconfigurator*. Either for the initial configuration of a system or for any change suggested, it must be checked whether it is possible to configure the system with the demanded features. This analysis operation is described by Benavides et al. [21] and an implementation that uses CSP solvers is proposed.

Finally, the initial configuration of the product will be defined by a selection of the non-core features that will be initially active. The *reconfigurator* component will be in charge of activating the selected features when the product is firstly launched.

Trinidad et al. approach has successfully been applied to generate an industrial real-time television. The system broadcasts a video composed by software, mixing TV signals, stored videos, Flash animations and any other kind of images or layers. Some kinds of effects can applied to the layers, such as black and white effect and lummakey and chromakey effects. Different user interfaces (UI) are needed to interact with the application. At least a basic UI that allows managing layers and effects

is required. Other UI are demanded to schedule TV compositions and to download SMS messages from a server and sending it to a Flash animation.

Trinidad et al. Approach - Mapping features intro components			
Scope	Multimedia systems: Industrial real-time television		
Variability Specification	Feature Model taking into account the cross-tree constrains (requires and excludes)		
Reconfiguration Infrastructure	<i>Feature componentes</i> with multiple interaces, <i>relationship components</i> to connect the former components and a <i>reconfigurator</i> to drive the dynamic behaviour.		
AC Adoption	Level 4	AC Relevance	Core
Reinforcement	Autonomous	Maturity	Stage 2

Table 3.6: Classification of Trinidad et al. Approach.

3.3.6 Mori et al. Approach

Mori et al. [112] (see Table 3.7) present a new approach where they address challenges in adaptive system construction and execution by combining certain aspect-oriented and model driven techniques. Models cope with complexity through abstractions and are used both to specify the dynamic variability at design time and to manage run time adaptations.

The variant models capture the variability of the adaptive application. The actual configurations of the application are built at run-time by selecting and composing appropriate variants. An adaptation model specifies which variant have to be selected depending on the context of the running application.

Specifically, they propose to model the variants instead of the configurations. Then, the configurations can then be built by automatically combining the variants. In practice this is achieved using Aspect-Oriented Modelling techniques for architecture models. Aspect oriented techniques are utilized to model the adaptation concerns separately from the other aspects of the system. The architecture models is a generic component model representing the main concepts needed to describe the topology of running systems: components, binding, ports, etc (see Figure 3.7 left).

Figure 3.7 right presents the conceptual model of the proposed approach. The approach is divided in two phases; design time and run-time. At design-time, the application base and variant architecture models are designed and the adaptation model is built. At run-time, the adaptation model is processed to produce the system configuration that should be executed.

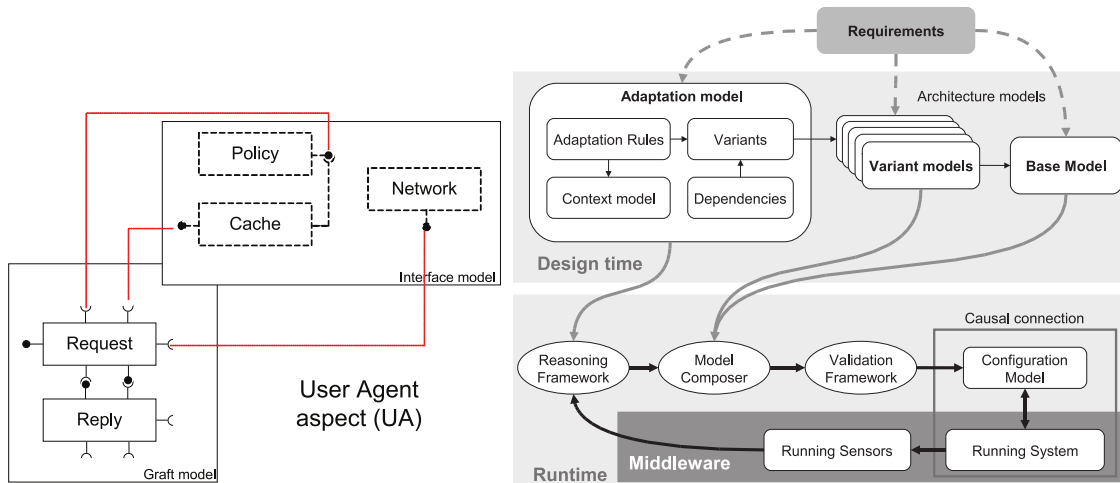


Figure 3.7: Mori et al. Approach

During run-time appropriate configurations of the application have to be built from the base and variant models. To select the appropriate configuration, the reasoning framework processes the adaptation model and makes a decision based on the current context. The output of the reasoning framework is one or more options that match the adaptation rules and satisfies the dependency constraints. For each of these options the complete model of the corresponding configuration can be built at run-time using model composition.

Because the idea of the approach is to build configurations on demand rather than enumerating all configurations, each new configuration has to be validated at run-time. The role of the validation framework is to process the configuration proposed by the reasoning framework in order to select the ones that are safe to deploy in the running system. The validation framework checks that the architecture model of the configuration is correct with respect to the constraints and protocols associated to the components it contains.

Once a configuration has been selected by the reasoning framework and checked

by the validation framework, it can be deployed in the running system. To ease the adaptation of the running system, a model representing the system at a higher level of abstraction is causally connected to it. This model is transformed to match the configuration that has been selected for adaptation. The running system is adapted thanks to the causal connection. Because the connection goes in both directions, it also allows checking that the system is actually running the required configuration.

Mori et al. Approach has been successfully applied in the context of mobile computing environments applications. These applications need to dynamically discover services from a wide range of options that may be unknown during design. However, their approach is applicable on many execution platforms since their models at run-time are provided as platform independent models.

Mori et al. Approach - Combining model driven and aspect oriented			
Scope	Dynamic service discovery for mobile applications		
Variability Specification	Architecture models and Aspect-oriented modelling techniques		
Reconfiguration Infrastructure	A reasoning framework decides on a set of aspects according to the new context, and a new configuration the application should adapt to is created by weaving these aspects. Then, reconfiguration commands are responsible for adding and/or removing bindings and/or components, etc.		
AC Adoption	Level 5	AC Relevance	Support
Reinforcement	Autonomic	Maturity	Stage 3

Table 3.7: Classification of Mori et al. Approach.

3.3.7 Hallsteinsen et al. (MUSIC) Approach

Hallsteinsen et al. [113] (see Table 3.8) argue that Dynamic Software Product Lines offers a suitable development model for developing configurations of many independently developed systems which use services from and provide services to each other.

They propose to combine (1) DSPL Architectures and (2) Service Level Agree-

ments (SLAs). SLA negotiation coordinates the configuration of a set of interacting systems by introducing service requests and service offers as a kind of dynamic variation point. This combination is realized in the MUSIC approach as follows.

1. **DSPL Architectures.** In the MUSIC framework the adaptation middleware monitors relevant context and resources. When significant changes occur it reconfigures the application to the configuration which has the highest utility in the new situation among the ones satisfying the resource constraints. Variation points are characterized by properties which vary between their variants (see Figure 3.8 left). These properties express functional and/or QoS properties of the provided service. The properties and the resource needs of an application variant are computed by predictor functions based on the properties and resource needs of the included component variants.
2. **SLA negotiation.** This negotiation enables late binding of services at run-time between a consumer and a provider. Through SLA negotiation, both the provider of the service and the conditions of the service usage may vary. Service bindings is a natural extension of the repertoire of variation points supported by the MUSIC framework and fits into the model. The discovered services with different offered SLs will be the variants that can be selected from and bound by SLA negotiation (see Figure 3.8 right). This is different from component variant binding since the client does not need to provide the resources to execute the service implementation.

Assuming a system of systems where all systems are built as DSPLs with utility and property predictor based decision models as described above, Hallsteinsen et al. propose that each system is configured separately but the configuration is coordinated through SLA negotiation.

In the system of systems, each system has a reasoner or planner working independently for local reconfiguration based on the decision models and run-time context. The planners also collaborate with each other to achieve best utilities by selecting and binding services via SLA negotiation. High level policies are applied to govern

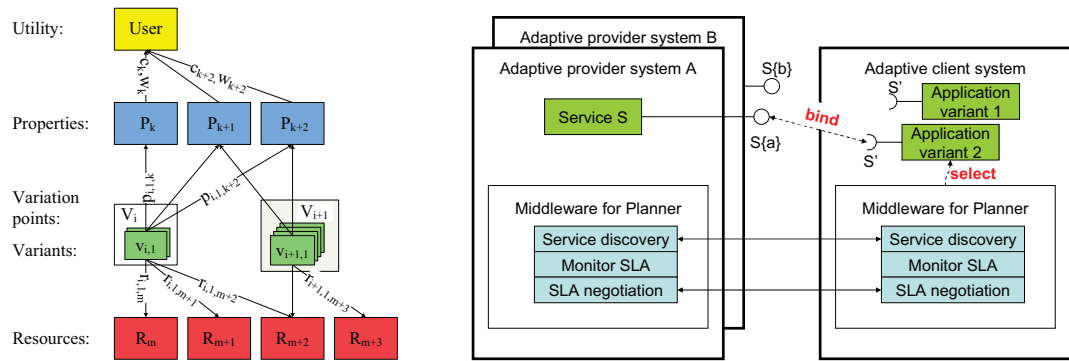


Figure 3.8: Hallsteinsen et al. Approach

the planner's decision, in particular, on adjusting the weights of the utility functions according to the run-time context.

Typically, a node is considered as a system. Each node can provide services to others (as a provider or server). At the same time, it can also use services provided by others (as a consumer or client). When a node is hosting a service, it will publish the service using specific service discovery technologies.

The client can discover published services based on service discovery mechanisms. Discovered services with published SLs are considered variants which the local planner can incorporate into the planning and adaptation process for local reconfiguration. Services of the same type with different providers are considered as different variants. In addition, if a service is published with optional SLs, each SL is considered to be a variant.

When a planner selects a service with a certain SL in a configuration, a negotiation process for the desired SL will be initiated with the corresponding service provider. If the negotiation process is successful, a SLA is established, and the service is provisioned and can be used with the SL agreed. If a SLA could not be created, the local planner will select another variant and enter into a similar negotiation process.

The MUSIC approach using the SLA negotiation introduced above has been validated in a sea monitoring case study. This case study involves a number of underwater sensor nodes taking measurements, and communicating these to a central measurement database, either through buoys, passing ships or autonomous under-

water vehicles (AUVs) which collect data from the sensors more or less regularly.

Hallsteinsen et al. Approach - MUSIC + SLA negotiation			
Scope	Service-Oriented Architectures: Sea monitoring case study		
Variability Specification	Conceptual model based on the notion of entities connected through ports and Service Level Agreements		
Reconfiguration Infrastructure	(1) Property annotations associated with ports and a utility function to determine the properties which matches a given context, and (2) Service Level Agreement negotiation		
AC Adoption	Level 5	AC Relevance	Support
Reinforcement	Autonomic	Maturity	Stage 2

Table 3.8: Classification of Hallsteinsen et al. Approach.

3.3.8 Parra et al. Approach

Parra et al. [114] (see Table 3.9) argue that using an SPL paradigm to build context-aware systems based on SOA services, enables a complete service development from requirements to implementation, and a management of context throughout the software lifecycle.

Specifically, Parra et al. propose an homogeneous Context-Aware Dynamic Service-Oriented Product Line (DSOPL) named CAPucine. Their goal is to define at the same time a service-oriented and context-aware product derivation that monitors the context evolution in order to dynamically integrate the appropriate assets in a running system. This target platform follows the service-oriented approach.

CAPucine is based on a model-driven approach. For every selected feature in a Feature model (see Figure 3.9 left), there is an associated asset that in CAPucine case, corresponds to a partial model of the product itself. Afterwards, CAPucine compose the selected partial models to have one integrated model that represents the product.

The next step is to transform this model to enrich it with concepts of the platform, and the implementation language. This is done by performing a series of model

to model transformations towards the platform and the implementation domains. Finally, the product is built by generating the code from the target domains.

The code generation produce the *context-aware assets* that can be integrated at run-time. The run-time integration depends on the environment state. *Context-aware assets* own the different alternative architectures of a system and their conditions of existence that depend on the environment state.

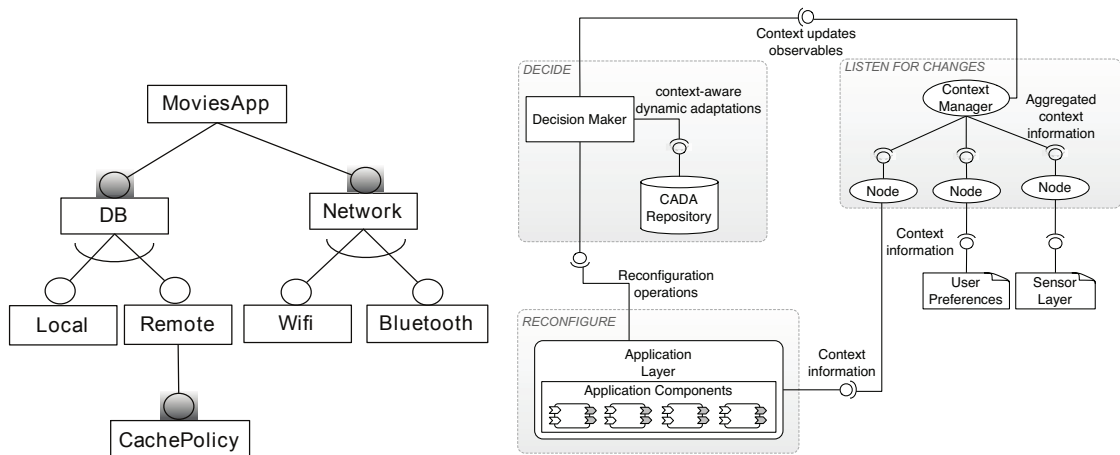


Figure 3.9: Parra et al. Approach

The context-aware assets include a definition of a context information that corresponds to the decision whether or not to adapt the system. Hence, CAPucine platform provides a context aggregation mechanism. Such a mechanism is in charge of getting the information from multiple sources and to provide a high-level view of information, so that, it can be evaluated.

The CAPucine platform is also able to suspend and resume the execution of the system, modify its structure by performing different operations like deploy, add, bind or delete components. This enables dynamic adaptation for each context-aware asset.

Overall, the platform architecture is depicted in Figure 3.9 right. The Context Manager element is composed of several nodes. Every node is in charge of recovering context information from different sources like a sensor layer who captures raw data from the environment, user preferences, and the Run-time Platform who provides information about current state and configuration of applications. Eventually, the Context Manager can also perform a processing of data, so that, it is presented

as single values which can be evaluated in the condition of each context-aware asset. The Decision Maker element is in charge of evaluating the context and decide whether or not to modify the application. It is linked to a repository of rules. The rules represent the clauses of each context-aware asset. Finally, the Run-time Platform element is where Application Components are executed. It controls the life cycle of all the application components and has access to their control mechanisms.

Parra et al. has applied CAPucine to an scenario which consists in a family of systems used to obtain and display information about movies. CAPucine is implemented using FraSCAti [115], an SCA platform with dynamic properties enabling binding and unbinding of components at run-time. Finally, for context sensing, CAPucine is based on COSMOS [116], which is a context-aware framework connected to the environment by the use of sensors.

Parra et al. Approach - CAPucine DSPL			
Scope	Service-Oriented Architectures: Movie System		
Variability Specification	Feature Model without taking into account the cross-tree constrains (requires and excludes)		
Reconfiguration Infrastructure	CAPucine platform which features: (1) COSMOS a contextaware framework connected to the environment by the use of sensors, and (2) FraSCAti, a Service Component Architecture with dynamic properties that enables to bind and unbind components at run-time		
AC Adoption	Level 5	AC Relevance	Core
Reinforcement	Autonomic	Maturity	Stage 2

Table 3.9: Classification of Parra et al. Approach.

3.3.9 Istoan et al. Approach

Istoan et al. [117] (see Table 3.10) argue that a convergence of Service Oriented Architecture (SOA) and Software Product Lines is highly possible. They suggest atomic services that are used to represent basic system features. A composition of such services creates a configuration, which is a product of the product line.

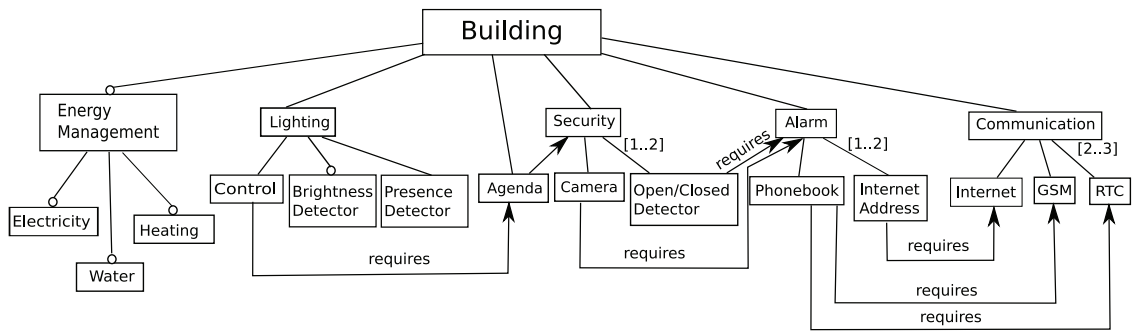


Figure 3.10: Istoan et al. Approach

To support this convergence of SOA and DSPLs, Istoan et al. present ENTIMID a service-based middleware designed to solve house automation issues such as devices interoperability, linkage facilities or scenarios descriptions. The aim of this middleware, is to offer a level-sufficient abstraction of inhouse devices, making it possible for high level services to interact with physical devices (such as lamps, heater or temperature sensors) and ease their management.

To introduce variability in ENTIMID, they capture the commonalities and variabilities among different configurations in terms of features (see Figure 3.10). In particular, they distinguish between two types of features: composite and atomic. Atomic features, present at the leaf level of the feature model, are directly mapped and implemented by existing services. For implementing a particular atomic feature, they choose between multiple existing services offering the same general functionality. Several atomic features are grouped together into a composite one. Such a composite feature may also include other composite features. Its role is to offer a new service to the user, not available before, whose functionality is derived from that of the atomic services it encompasses. Finally, a product of our SPL, called a configuration, contains one or more composite services.

Variability in ENTIMID requires a particular restriction on the feature modelling technique: a configuration will be ultimately decomposed, at the leaf level of the feature models. That is, only leaf features can represent services. These leaf features are implemented as atomic services. The choice of a particular service for a given feature depends on service availability, quality of service requirements and user preferences

The different configurations are assembled from the existing atomic services by the ENTIMID platform. ENTIMID offers possible methods and facilities for monitoring the state of the system, determining services that need to be replaced and performing the actual replacement under the appropriate conditions.

Istoan et al. plan to apply the ENTIMID platform extended with run-time reconfiguration capabilities to the building automation domain. At the moment of writing this thesis, the extended ENTIMID platform is work on progress and it lacks an implementation of the reconfiguration infrastructure. Istoan et al. plan to implement this infrastructure by means of MDD techniques using Kermeta [118].

Istoan et al. Approach - Extended ENTIMID with DSPL Architecture			
Scope	Service-Oriented Architectures: Home automation		
Variability Specification	Feature Model with the restriction of only leaf features can represent services		
Reconfiguration Infrastructure	The extended ENTIMID platform is work on progress and it lacks an implementation. The platform is planed to be implemented by means of model driven techniques using Kermeta.		
AC Adoption	Level 4	AC Relevance	Support
Reinforcement	Autonomous	Maturity	Stage 1

Table 3.10: Classification of Istoan et al. Approach.

3.4 Discussion

Inspired by biology, autonomic computing has evolved as a discipline to create software systems and applications that self-manage in a bid to overcome the complexities and inability to effectively maintain current and emerging systems. To this end, the presented approaches enable the production of systems which adhere to different degrees of Autonomicity.

We have compiled a graphic representation (see Figure 3.11) in which we place the approaches presented in this Chapter. These approaches are categorised by the Maturity of their methodology (horizontally) and the level of autonomic computing

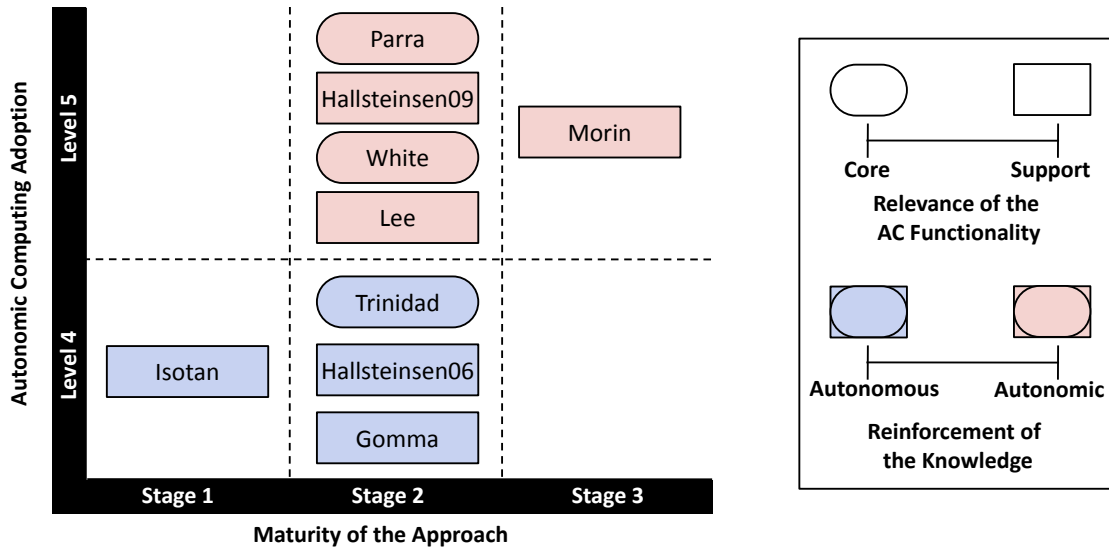


Figure 3.11: Classification of the Approaches: Maturity - Autonomic Level

adoption (vertically). In addition, the graphic representation of the approaches denotes their type of Reinforcement of the Knowledge (shape), and the relevance of the Autonomic Computing functionality (colour). While some of the choices are highly subjective, we have tried to place them in a category that highlights the major contribution of the approach.

According to the classification of the approaches presented in this Chapter, we can claim that there are many open challenges which are receiving less attention at the point in time we write this thesis as follows:

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

Challenge 1 Develop guidelines, techniques and tools to support engineers from system design to execution.

- Only Mori et al. approach [112] is at beginning of Stage 3, but they focuses on reducing the number of configurations that need to be considered and their

approach (as the approaches of Stage 2 and 1) lacks support for other notable concerns such as validation analysis, debugging or tracking capabilities.

Challenge 2 Further studies about how to address concerns such as validation analysis, debugging or tracking capabilities.

- Furthermore, all of the analysed approaches are also missing (and we believe the community should be considering) the evaluation of the safety and reliability of run-time reconfigurations. These properties are essential for the development of reliable reconfigurable systems.

Challenge 3 Carry to successful evidences about of the safety and reliability of run-time reconfigurations.

Overall, we believe it is indispensable to come to a seamless software engineering approach which supports autonomic system engineers from design time to run-time in order to address the former open challenges.

3.4.1 Architectural Patterns

According to the approaches presented in this Chapter, we have also identified interesting patterns in the reconfiguration architectures presented in this chapter. Specifically, these reconfiguration architectures can be classified into two categories according to the way in which system reconfiguration is considered: (1) Connected DSPL (the DSPL is in charge of the product reconfiguration), and (2) Disconnected DSPL (The system itself is in charge of the product reconfiguration).

Connected DSPLs stay in touch with systems in order to send them updates. These updates enable systems to deal with context changes. Figure 3.12 shows the steps to send the updates from the DSPL to the systems.

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

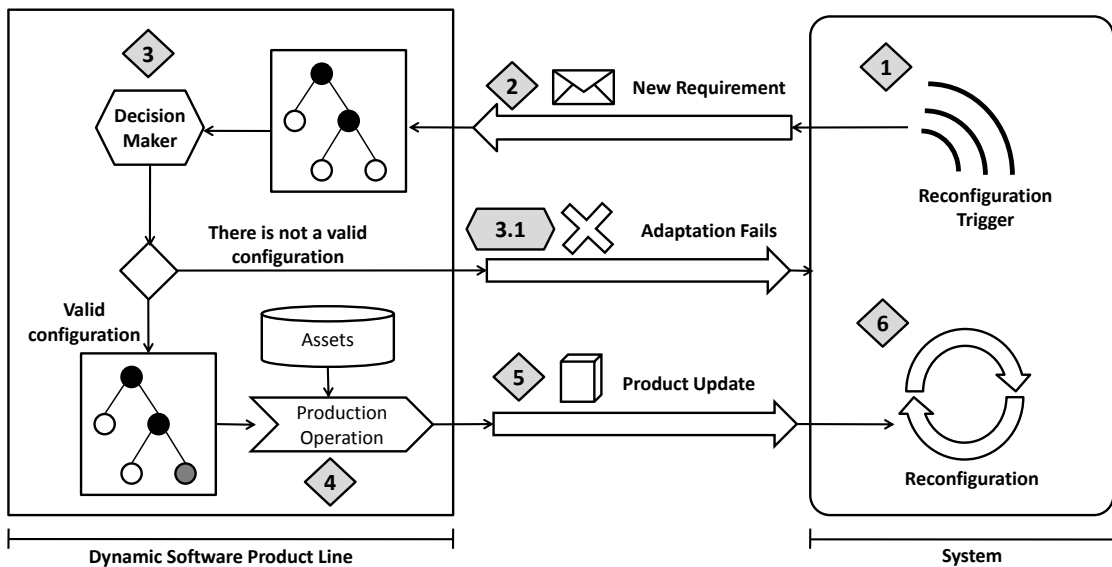


Figure 3.12: Connected DSPL Overview

2. The system sends information about the change to the SPL. Optionally, the system 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 system variant.
 - (a) If there is no variant that satisfies the product requisites, then the SPL notifies the system and the reconfiguration process fails.
4. The SPL generates the system 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 system.
6. The system updates itself using the update information from the SPL.

The main characteristics of this pattern for reconfiguration infrastructures are as follows:

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

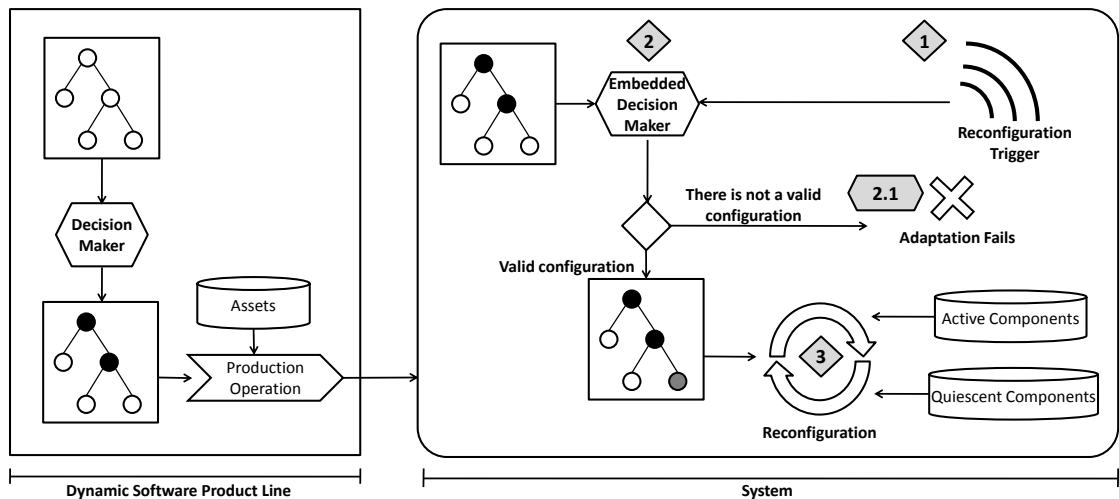


Figure 3.13: Disconnected DSPL Overview

- 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 system. If this connection becomes unavailable then the adaptation cannot be performed.
- Computational overload.** An disconnected DSPL approach introduces the following additional overload in the system execution: (1) the communication with the SPL (to get system updates) and (2) the on-line installation of updates.

Disconnected DSPLs produce systems which can reconfigure itself to deal with contextual changes in a autonomic manner. Compared with connected DSPLs, the system reconfigures itself without any DSPL interaction. Specifically, systems are augmented with variability knowledge and extra components in order to perform the reconfiguration as Figure 3.13 shows.

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

2. The system calculates a new configuration to deal with the sensed change.
 - (a) If there is no configuration that satisfies the product requisites, then the reconfiguration process fails.
3. The system reconfigures itself to apply the calculated configuration. The reconfiguration operation implies (1) start/stop components and (2) establish/destroy connections between them.

The main characteristics of this pattern for reconfiguration infrastructures are as follows.

- **Autonomic degree.** The system has no dependency of the SPL to perform the reconfiguration because there is no connection required between the SPL and the system. The reconfiguration only depends on the system resources.
- **Reconfiguration capabilities.** In general, the more variability knowledge the system has about itself, 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 extra system components. Some components conform the initial system configuration, while others are used in system reconfiguration. Therefore, 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 reconfiguration is triggered. This overload comes from (1) the variability queries and (2) the execution of the reconfiguration (starting stopping and linking system components).

3.5 Conclusions

At the time of writing this thesis, it has been five years since first attempts to address run-time reconfiguration of system families, and we are now beginning to observe concentrations of research emerging in key application domains. According

to the approaches presented in this chapter, the main application domains are mobile devices and Service Oriented Architectures. In addition, there are also valuable contributions in other domains such as: system automation, smart homes and multimedia services. Figure 3.14 classifies the presented approaches by the application domains (horizontally) and the reconfiguration infrastructure type (vertically).

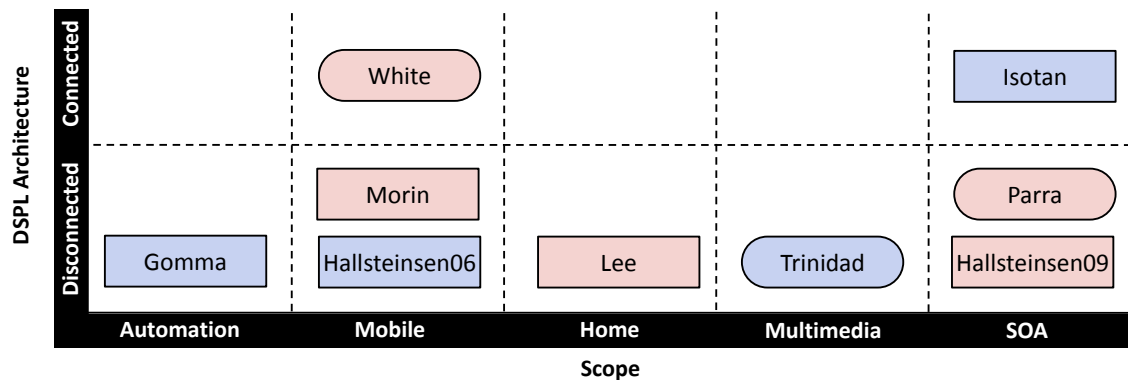


Figure 3.14: Classification of DSPL: Scope - Infrastructure

Overall, the above application domains concern the building of intelligent environments from a number of, potentially, heterogeneous devices such as sensor nodes or mobile devices. The complexity of installing and maintaining such a system and keeping it running in a robust way, lends the approaches to achieve autonomic computing.

Although autonomic computing has become increasingly interesting and popular it remains a relatively immature topic its achievement in a systematic manner. However, as more approaches become involved, the established research can be reuse, adding to the maturity of the area. Furthermore, we believe that in the future the topic will become integrated into general SPL and MDD communities and not be seen as the separate area it is today (DSPL community).

Chapter 4. OVERVIEW OF THE APPROACH

“If we knew what we were doing, it wouldn’t be called research, would it?”

– Albert Einstein (1879-1955).

4.1 Overview of the Chapter

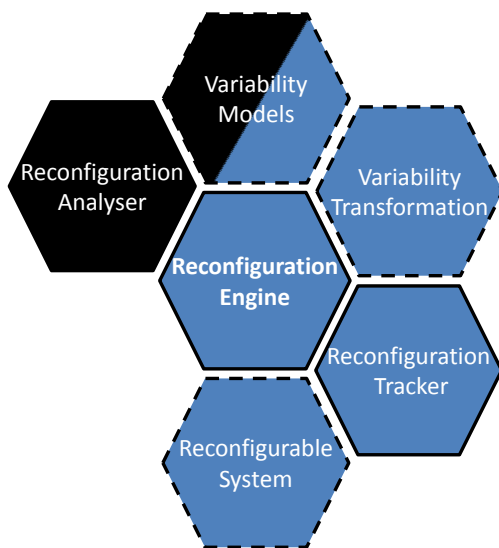


Figure 4.1: Scope of Chapter 4

Autonomic Computing transfers maintenance responsibilities to the software itself. By automating tasks such as installation, healing or updating, system operation is simplified at the expense of increasing its internal complexity. Our work provides an approach for the development of autonomic systems through the use of variability models at run-time. The purpose of these models at run-time is that variability models can be used to provide a richer semantic base for run-time decision-making related to autonomic behaviour. Based on the widespread modelling tools of the *Eclipse Modelling* project, a software infrastructure has been developed to support our approach. This chapter briefly introduces an overview of this approach, which is developed in the next Chapters. Then, we identify and overview the main building blocks of the approach. We also propose a process to apply this approach. Finally, we show how the approach has been put into practice and evaluated throughout the case study of a Smart Hotel.

4.2 Introduction

Previous studies have highlighted that people continuously reconfigure domestic spaces and the technologies involved in order to support their activities [19]. In order to reduce this configuration effort, the following autonomic capabilities can be provided:

Self-configuring. New kinds of devices can be incorporated to the system. For example, when a new movement/presence detector is added to a home location, the different smart home services such as security or lighting control should automatically make use of it without requiring configuration actions from the user.

Self-healing. When a device is removed or fails, the system should adapt itself in order to offer its services using alternative components to reduce the impact of the loss of the device. For example, if an alarm fails, the Smart Home can make the home lights blink as a replacement for the failed alarm.

Self-adaptation. The needs of users are different and change over time. The system should adjust its services in order to fulfill user preferences. For example, when all users leave home, services in the home should be reorganized to give priority to security.

We consider this autonomic behaviour to be closely related to context adaptation. Context adaptation is a system's capability to gather information about the domain that it shares an interface with, to evaluate this information and to change its observable behavior according to the current situation [119]. The individual capabilities of autonomic systems (i.e., self-configuring, self-healing and self-adaptation) also require the system to infer knowledge from the current situation and to trigger an appropriate response. However, autonomic computing places the emphasis on freeing system users from the details of system operation and maintenance and on providing users with systems that run 24/7 [26].

To achieve this autonomic behaviour, we argue to leverage the models produced as artifacts from Model Driven Engineering (MDE) methodologies as if they were the policies that drive the autonomic behaviour of the system at run-time. In MDE, a model is an abstraction or reduced representation of a system that is built for

specific purposes. For example, technology-independent models of software describe systems using concepts that abstract over the underlying computing technologies. We share this view of what constitutes a model and explore the use of models at run-time to drive the autonomic behaviour of the system. Our decision to use models at run-time to achieve Autonomic computing comes for two reasons.

- If the model reflects the system architecture and its operational context, then the model can provide up-to-date and exact information to drive subsequent adaptation decisions.
- If the model is system-connected¹, then adaptations can be made at the model level rather than at the system level.

That is, under the assumption that the model correctly mirrors the managed system, the model can be used to verify that system integrity is preserved when applying an adaptation, i.e. we can guarantee that the system will continue to operate correctly after the planned adaptation has been executed. This is because changes are planned and applied to the model first, which will show the state of the system resulting from the adaptation, including any violations of constraints or requirements of the system present in the model. If the new state of the system is acceptable, the plan can then be effected onto the actual managed system, thus ensuring that the model and implementation are consistent with respect to each other.

In particular, our approach makes use of (1) Variability Models [20] and (2) Dynamic Product Line Architectures [18]. Variability models specify the possible configurations of a Smart Home, while a Dynamic Product Line Architecture can be rapidly retargeted to a specific configuration. Below, we provide a brief overview of Variability Modelling and a Dynamic Product Line Architecture.

Variability Modelling: From the different techniques that are suited for variability analysis, we have chosen feature-based model languages. Feature modelling

¹The model is linked in such a way that it constantly mirrors the system; if the system changes, the model must also change, and vice versa.

is widely used for the specification of system functionality in a coarse-grained fashion by means of the feature concept (an increment in system functionality). The features are hierarchically linked in a tree-like structure through variability relationships such as optional, mandatory, single-choice and multiple-choice. Some of the features denote the initial system configuration, while the other features represent potential variants since they may be activated in the future.

Dynamic Product Line Architecture: In order to allow a flexible reconfiguration, we have considered the architecture of a Dynamic Product Line. This architecture is based on different components and their communication channels. We classify these components into two categories: Services and Devices. This architecture allows an easy reconfiguration since communication channels can be established dynamically between the components, and these components can dynamically appear or disappear from configurations.

Our research shows that these variability models can be used at run-time to assist the system in determining the steps that are necessary to reconfigure itself. In particular, we argue that a system can activate/deactivate its own features dynamically at run-time according to the fulfilment of Context Conditions.

In order to turn into reality the proposal, a Model-based Reconfiguration Engine (named MoRE) was developed. MoRE enables the symbiosis between *Variability Modelling* at run-time and *Dynamic Product Line Architectures* to pay dividends in the field of Autonomic Computing. Specifically, MoRE implements the model operations to management models at run-time. These operations are in charge of determining how the system should evolve and the mechanisms for modifying the system architecture accordingly. Thus, systems make use of the knowledge captured by variability models as if they were the policies that drive the autonomic evolution of the system at run-time.

For validation purposes, the current proposal has been applied to the Smart Home domain (see www.autonomic-homes.com). We have selected the Smart Home domain because AC capabilities can address some of the adaptation and reconfigurations challenges of this domain [19]. First, because of its nature as a shared

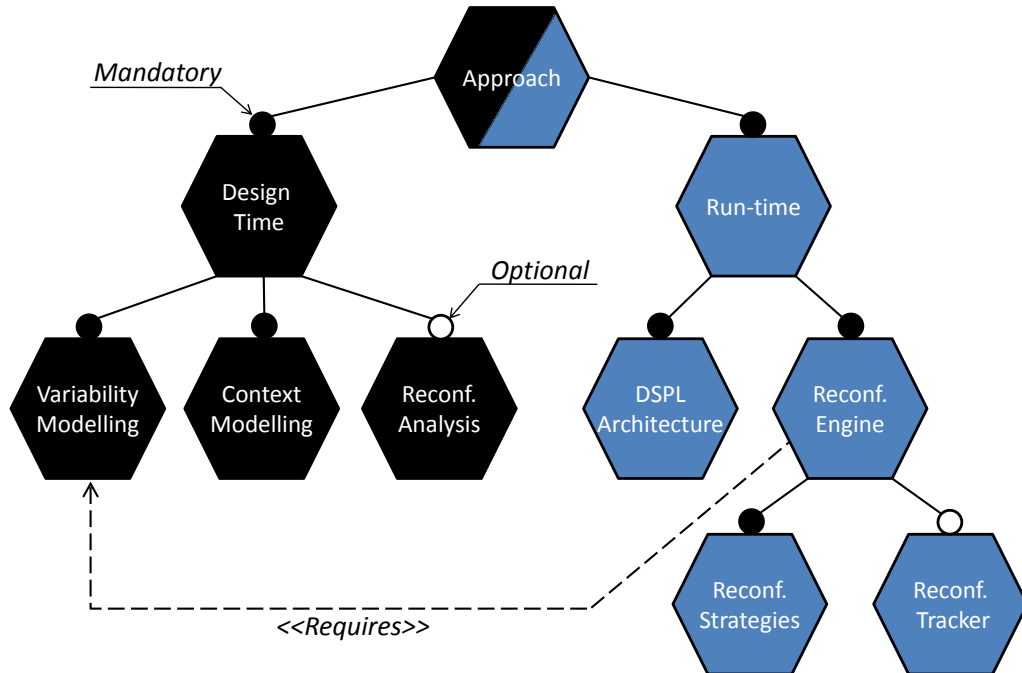


Figure 4.2: Main Building Blocks of the Approach

environment, different users use the same room over time. Each user has its own preferences for the room, which should be adjusted to improve the quality of their stay; second, the preferences of the users change depending on the activity performed (e.g., the users usually have different preferences when they are watching a movie than when they are working).

4.3 Main Building Blocks

Figure 4.2 presents the main building blocks of the proposed approach. Each building block is denoted by an hexagon and these building blocks are related to the approach by mandatory, optional and requires relationships. From a methodological perspective the approach is divided in two phases; design time (black hexagons) and run-time (blue hexagons). At design-time, the models that specify the system variability and the system context are built. At run-time, these models are queried in response to context events to produce the system reconfiguration that should be executed. The main building blocks of the approach are:

- Design time building blocks.

- **Variability Modelling.** Variability models enable us to describe the variants in which a system can evolve. We argue that in response to changes in the context, the system itself can query these variability models in order to determine the necessary modifications to its architecture. Regarding Variability Modelling techniques, we have successfully applied our work to both Feature Models [120] and Common Variability Language specifications [121].
 - **Context Modelling.** For context modelling, we use an ontology-based context model that leverages Semantic Web technology and OWL (Web Ontology Language) [122]. Given such ontology, we define context conditions as queries that check for values in the system context ontology. The fulfillment of these conditions triggers the system reconfiguration.
 - **Reconfiguration Analysis.** For dependable systems, it is indispensable to have a means to analyze the reconfigurations before performing them. To address the above problem, our approach validates the configurations resulting from the simultaneous fulfillment of context events at design time. Therefore, unexpected configurations can be avoided. In particular, we analyse Variability Models by means of the FAMA framework [21] for variability analysis.
- Run-time building blocks.
 - **DSPL Architecture.** The Reconfigurable architecture of a DSPL promotes that each architecture component is designed to be capable of transitioning to a state where it can be reconfigured. Under a set of reconfiguration commands, the components that make up the architecture dynamically cooperate to change the configuration of the architecture to a new configuration.
 - **Reconfiguration Engine.** To enable autonomic behaviour, the system must evolve from one configuration to another by itself. Since the reconfiguration in our approach is driven by variability models at run-time, a Model-based Reconfiguration Engine (MoRE) is provided to address

context changes. MoRE uses the variability models to determine how the system should move from a consistent architecture to another consistent architecture by means of reconfiguration actions. These reconfiguration actions modify the system components accordingly.

- * **Reconfiguration Strategies.** The model operations which manage the run-time variability models are grouped in a so called variability transformation. We propose a set of alternative strategies for implementing this variability transformation. These strategies implement the same reconfiguration functionality but they have different extra-functional properties. For instance, they do not offer the same performances. These strategies enable engineers to set up MoRE with the most suitable strategy for each particular concern such as debugging or performance.
- * **Reconfiguration Tracker.** Given a system reconfiguration and the variability model, the Reconfiguration Tracker records a run of the reconfiguration at the abstraction level that the variability model induces. These trace entries provide a way to formally and quantitatively characterize and investigate the concrete reconfiguration the trace was generated from, and also the overall running of the system.

The above building blocks that leverage software models extends the applicability of MDE techniques to the run-time environment, blurring the line between development models and run-time models.

4.4 Application

To achieve autonomic computing, we have used the above building blocks to provide support to autonomic system engineers from system design to execution. At design time (see top of Figure 4.3), we take advantage of current variability and context modelling techniques in order to specify the context and architecture of the system, and how the system architecture can be adapted to manage context changes. Furthermore, this stage also benefits from the whole range of typical gains brought by

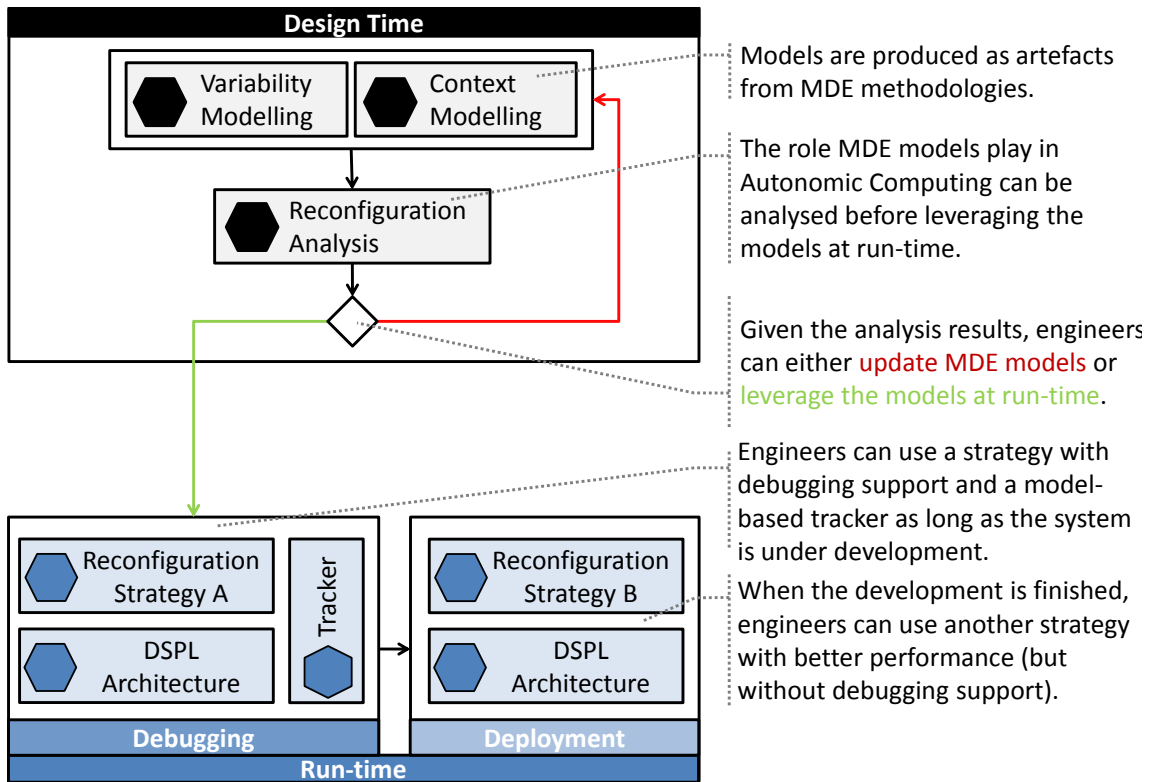


Figure 4.3: Simplified overview of the Process to Apply the Approach

MDE approaches (i.e. validation, verification, reuse and automation). In fact, we also take advantage of current techniques for variability analysis in order to conduct a thorough analysis of the models for the purpose of validation.

At run-time (see bottom of Figure 4.3), the design knowledge and existing model-based technologies can be used to support Autonomic Computing. In this way, the modelling effort made at design time is not only useful for producing the system but also for providing autonomic behaviour during execution. This stage covers both debugging and deployment and it involves the building blocks of DSPL Architecture, Reconfiguration Strategy and Reconfiguration tracker.

Figure 4.4 presents an overview of the process to apply the approach. Specifically, this process features six tasks. For each one of these tasks, we provide the following information: name of the task, a brief description, involved Building Blocks and tool support.

1. **Task.** To specify the variability of the reconfigurable system.

Description. We propose to guide the design of a reconfigurable system

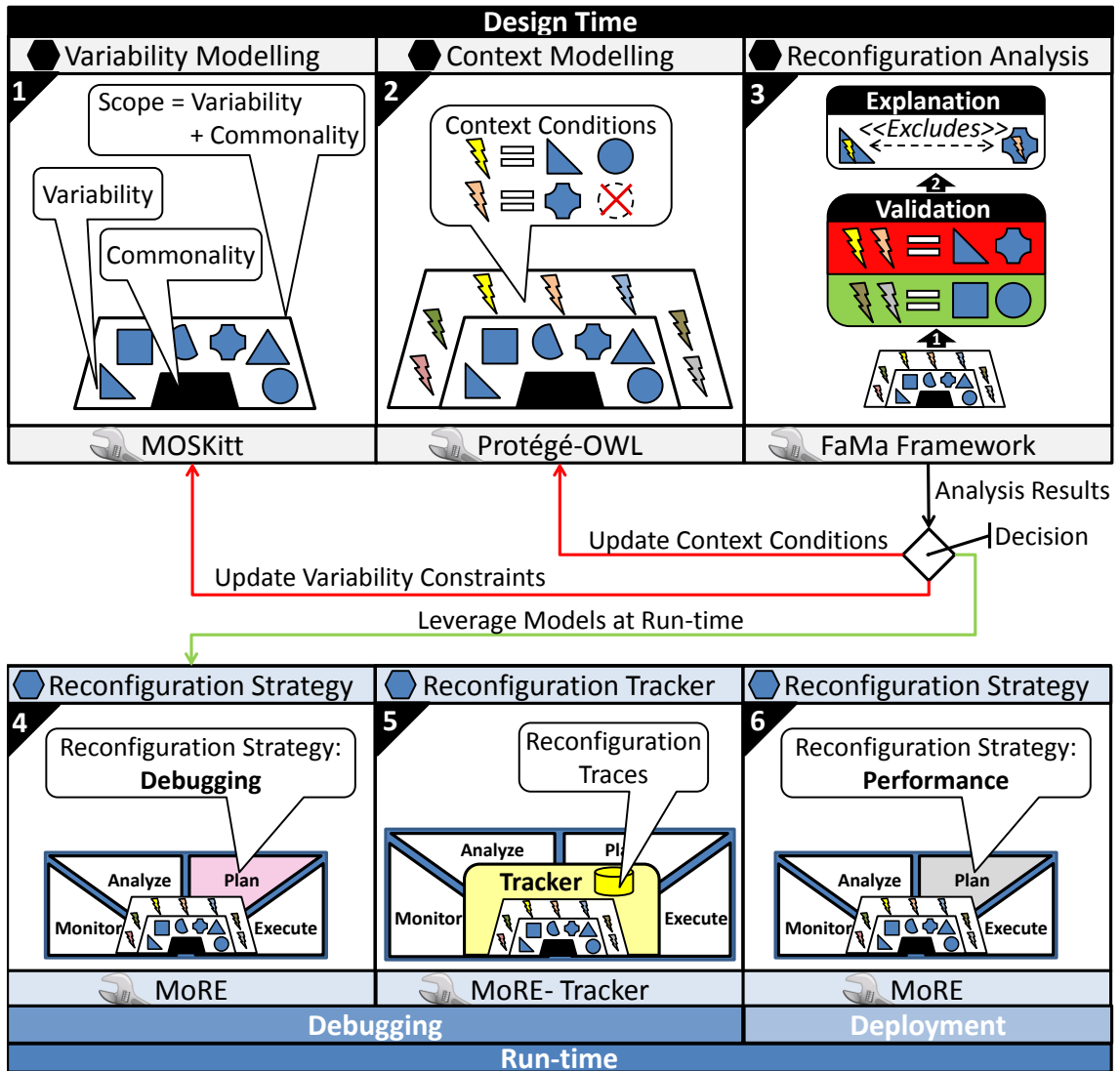


Figure 4.4: Overview of the Process to Apply the Approach

by scope, commonality, and variability (SCV) analysis [20]. SCV captures key characteristics of the reconfigurable system, including its (1) scope, which defines the domain of the system, (2) commonalities, which describe the attributes that come up across all feasible configurations of the system, and (3) variabilities, which describe the attributes unique to the different configurations of the system.

Involved Building Blocks. Variability Modelling.

Tool Support. MOSKitt² is a free Modelling platform, built on Eclipse which

²<http://www.moskitt.org/eng/moskitt0/>

is being developed by the Valencian Regional Ministry of Infrastructure and Transport. This modelling platform provides editors for several modelling languages such as Feature models or PervML (a DSL for Smart Homes), as well as code generation capabilities.

2. **Task.** To specify the context of the reconfigurable system.

Description. The context of the reconfigurable systems is specified by means of the OWL language. This language provides a vocabulary for describing system context knowledge, and for specifying conditions in the context. The fulfillment of these context conditions triggers a set of changes in the variants that conform the system configuration.

Involved Building Blocks. Context Modelling.

Tool Support. Protege-OWL is a free open source ontology editor and knowledge-base framework. An OWL ontology may include descriptions of classes, properties and their instances. Given such an ontology, the knowledge-base framework specifies how to derive its logical consequences, i.e. facts not literally present in the ontology, but entailed by the ontology instances.

3. **Task.** To analyze the reconfigurations before performing them.

Description. The configurations resulting from the simultaneous fulfillment of context conditions are validated at design time. This enables us not only to obtain a valid-invalid tag for each configuration, but also to know the reasons why a particular configuration is invalid. Given this information, we can update either the variability constraints or the context conditions to achieve a invalid-configurations free specification that can be used at run-time.

Involved Building Blocks. Reconfiguration Analysis.

Tool Support. FaMa is a Framework for automated analysis of feature models that integrates some of the most commonly used logic representations and solvers proposed in the literature. This framework enables to determine if a system configuration is valid (according to variability constraints), and it can also provide explanations about invalid configurations.

4. **Task.** To debug the run-time reconfigurations.

Description. Given the fact that not all potential run-time failures can be anticipated during system design [123], we can set up MoRE with a debugging-enabled reconfiguration strategy. This strategy keeps the history of system configurations. Therefore, we should use this strategy as long as the system is under development.

Involved Building Blocks. DSPL Architecture, Reconfiguration Engine and Reconfiguration Strategies.

Tool Support. MoRE featuring a debugging-enabled reconfiguration strategy.

5. **Task.** To Keep Track of the Reconfigurations.

Description. In the context of experimentation, MoRE can store trace entries about the reconfigurations. This provide us with information for a poster analysis, which ranges from context conditions to reconfiguration plans.

Involved Building Blocks. DSPL Architecture, Reconfiguration Engine and Reconfiguration Tracker.

Tool Support. MoRE featuring the Reconfiguration Tracker.

6. **Task.** To deploy the system in the target platform.

Description. Once the development is finished, we are not interested in debugging information any longer. Therefore, we can set up MoRE with another reconfiguration strategy which lacks debugging support but achieves better performance.

Involved Building Blocks. DSPL Architecture, Reconfiguration Engine and Reconfiguration Strategies.

Tool Support. MoRE featuring a debugging-enabled reconfiguration strategy.

Although some of the steps that conform the process to apply the approach can be skipped (for instance, variability analysis or reconfiguration debugging), we strongly recommend to perform all of the them. The whole process (as is proposed in this section) is conceived to achieve a system free of unexpected reconfigurations at run-time.

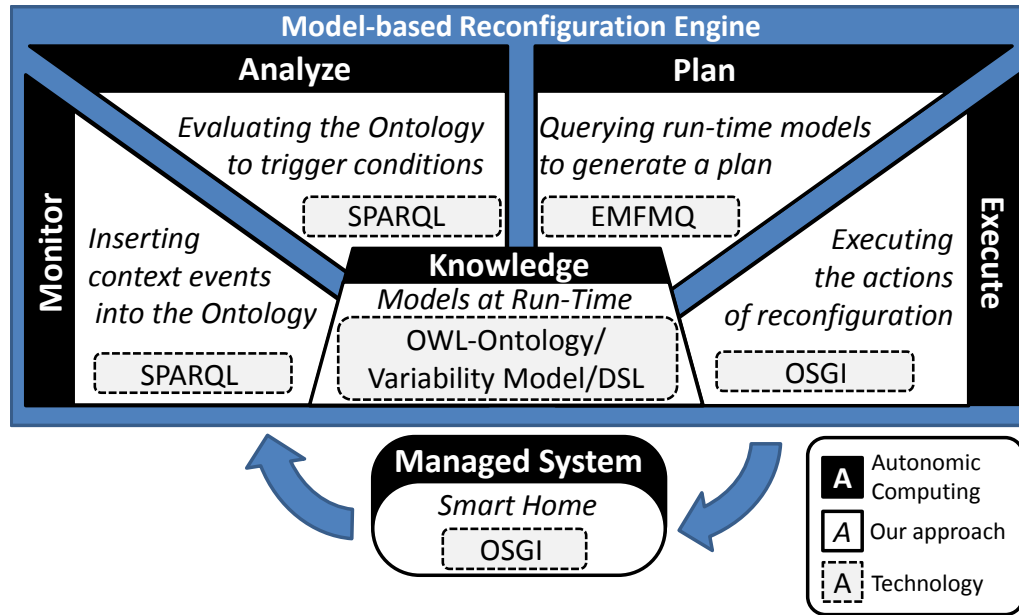


Figure 4.5: Overview of the Run-time Reconfiguration of the Approach

4.5 Implementation

To achieve autonomic computing, we also provide an execution platform for variability models at run-time. In particular, we have developed a model-based version of the IBM reference model for autonomic control, which is called the MAPE (Monitor, Analyse, Plan, Execute) loop (see Chapter 2 for a detailed description). The overall reconfiguration steps are outlined in Figure 4.5. A Context Monitor uses the run-time state as input to check context conditions. If any of these conditions is fulfilled, then MoRE queries the run-time models about the necessary modifications to the architecture. Given the model response, MoRE elaborates Reconfiguration Actions which modify the system architecture and maintain the consistency between the models and the architecture.

The above model-based version of IBM's reference model for autonomic control makes an intensive use of models. Context events and system variability are represented by models. Context events are represented by means of OWL ontologies, and system variability is captured by means of variability models. For performing the system reconfiguration, information is extracted from these models. Different model query technologies are used at run-time by MoRE depending on the modes involved.

MoRE uses SPARQL for OWL manipulation and Eclipse Model Query (EMFMQ) for variability model manipulation. Next, we briefly present both SPARQL and EMFMQ technologies.

- **SPARQL** is the W3C recommendation query language for RDF triples. This query language is based on graph-matching techniques. Given a data source, a query consists of a pattern which is matched against the data source, and the values obtained from this matching are processed to give the answer. The data source to be queried can be an OWL model as is the one of our ontology for system context.
- **EMFMQ** provides an API to construct and execute query statements in a SQLlike fashion. These query statements can be used for discovering and modifying model elements. Queries are first constructed with their query clauses and then they are ready to be executed.

Finally, the reconfiguration of the system is performed by executing reconfiguration actions that deal with the activation/deactivation of components and the creation/destruction of channels among components. Although our general approach is not platform-dependent, we take advantage of the concrete platform to implement the reconfiguration actions. MoRE makes use of the OSGi framework [124] for implementing the reconfiguration actions by means of the the OSGi capabilities to install, start, restart and uninstall components without having to restart the entire system.

4.6 Validation

The presented work has been validated from three different perspectives: (1) Scalability of the approach, (2) reliability-based risk of run-time reconfigurations and (3) degree of autonomic behavior achieved as follows.

1. **Scalability of the approach.** The introduced model-based reconfiguration is still subject to the same efficiency requirements as the rest of the system

because the execution of the reconfiguration impacts the overall system performance. Therefore, we were interested in analyzing to what extent system performance could be affected using complex models at run-time. Experimentation results show that our approach gathers the necessary knowledge from the run-time models to perform the reconfiguration without drastically affecting the system response.

2. **Reliability-based risk of run-time reconfigurations.** The presented approach encompass systems that are capable of modifying their own behavior with respect to changes in their operating environment by using run-time reconfigurations. However, a failure in these reconfigurations can directly impact the user experience. Thus, we were concerned with reliability-based risk of run-time reconfigurations, which depends on both the probability that the software product will fail in the operational environment (availability) and the consequences of malfunctioning (severity). Experimentation revealed that the reconfigurations achieved a high level of reliability
3. **Degree of autonomous behavior achieved.** To determine the level of autonomous behaviour that can be achieved with our proposal, we have obtained theoretical results about the autonomous behaviour specified by Feature Models at run-time. Furthermore, we have also asked users whether or not they considered the system reaction to be adequate taking into account the defined context events. Acceptance for the reconfiguration scenarios was high. Most of the users considered the behaviour provided to be a good response to the context events.

To evaluate the above concerns, we have developed a Smart Hotel case study (see Appendix A) following the guidelines for case study research by Runeson and Höst [125]. The Smart Hotel reconfigures its services according to changes in the surrounding context. In particular, a hotel room changes its features depending on users' activities to make their stay as pleasant as possible.

This case study was deployed with real devices (EIB-KNX and RFID), and it was performed with the participation of human subjects. Two major challenges were

identified and addressed with the involvement of human subjects in this reconfiguration evaluation.

- Run-time reconfigurations are triggered by context events, many of which are difficult to be reproduced in practice (e.g., a fire starts). To successfully evaluate reconfigurations, we must enable participants to trigger those reconfigurations that are relevant for the experimentation, not only those reconfigurations that can be easily triggered.
- When reconfigurations are performed some of the effects can be easily perceived (e.g., an alarm is triggered) while others are not (e.g., some sensors are deactivated). To successfully evaluate reconfigurations, we must enable participants to understand and evaluate the effects of reconfigurations. If participants misunderstand reconfiguration effects, they will not be able to evaluate the system response.

Overall, the evaluation of the case study revealed positive results that can encourage researchers and practitioners to apply model-based run-time reconfigurations to other promising areas of research such as mobile devices or automotive systems. However, the participant feedback in this study highlights issues with recovery from a failed reconfiguration or a reconfiguration triggered by mistake. To address these issues, we also provided some guidelines learned in the case study.

Finally, we conclude that the approach achieved satisfactory results regarding reliability-based risk; nevertheless, we must provide users with more control over the reconfigurations or the users will not be comfortable with reconfigurations even though they achieve a high level of reliability.

4.7 Conclusions

Autonomic Computing plays a key role in simplifying the use of systems by reducing the need for maintenance. We see models at run-time as an important contribution to the field of autonomic computing providing metainformation to drive autonomic decision making. This is done by means of a planned reutilization of the

efforts invested at design time. The benefits are immediate, as the design knowledge and existing model-based technologies can be reused at runtime. The runtime variability models support the autonomic behaviour of systems when triggered by changes in the environment. We have applied this approach to an application in the smart-homes domain, obtaining valuable validation of the approach.

Chapter 5. AUTONOMIC COMPUTING THROUGH THE USE OF VARIABILITY MODELS

“The art of progress is to preserve order amid change, and to preserve change amid order.”

– Alfred North Whitehead (1861-1947).

5.1 Overview of the Chapter

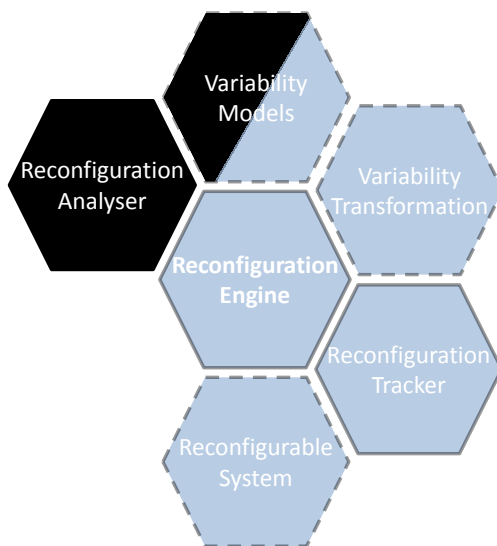


Figure 5.1: Scope of Chapter 5

Variability models enable us to specify not only current features of a system but also potential features since they may be activated in the future. We argue that in response to changes in the context, the system itself can query these variability models in order to determine the necessary modifications to its architecture. For instance, a smart home system can trigger the activation of both *In Home Detection* and *Occupancy Simulation* features when all the inhabitants leave the home.

First, this chapter presents variability modelling in the context of MDD-SPLs. Then, we argue how the modelling effort made at the MDD-SPL is not only useful for producing the system but also for

providing autonomic behaviour during execution. The knowledge previously captured in variability models is used to describe the variants in which a system can evolve.

Second, to determine the level of autonomic behaviour that can be achieved with our approach, we have obtained theoretical results about the autonomic behaviour specified by the variability Models. Specifically, we make use of a state machine for this purpose since, in practice, engineers use state machines to represent and check adaptation policies [112].

Third, since the Variability Models, which determine the autonomic behaviour, are available at design time, we are able to conduct a thorough analysis of the specifications for the purpose of validation. We are able to guarantee deterministic reconfigurations at run-time, which is essential for reliable systems.

Finally, we have validated our work using two different variability modelling techniques: Feature Models and the Common Variability Language (CVL) [121]. At [126], we have successfully applied our approach using CVL.

5.2 Variability Modelling

Variability modelling is regarded as the enabling technology for delivering a wide variety of software systems in a consistent and comprehensive way. The key is to build a base on the *commonalities* and efficiently express and manage the *variability* of the systems. According to [20], a *commonality* is an assumption held uniformly across a given set of systems. Frequently, such assumptions are components with the same specification for all the systems. Conversely, a *variability* is an assumption true in only some the systems, such as a component with different specification for at least two systems.

5.2.1 Variability and Software Product Lines

Variability modelling is closely related with product lines. Software Product Lines refer to methods, tools and techniques for creating and maintaining a collection of similar software systems from a shared set of software assets. Software product lines

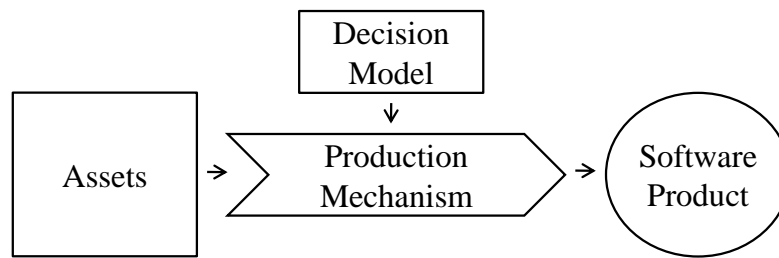


Figure 5.2: SPL main concepts

can be described in terms of four concepts, as illustrated in Figure 5.2.

- **Software asset inputs.** A collection of software assets (such as requirements, source code components, test cases, architecture, and documentation) that can be configured and composed in different ways to create all of the products in a product line. Each of the assets has a well defined role within a common architecture for the product line. To accommodate variation among the products, some of the assets may be optional and some of the assets may have internal variation points that can be configured in different ways to provide different behavior.
- **Decision Model.** Decisions describe optional and variable features for the products in the product line. Each product in the product line is uniquely defined by its product decisions (choices for each of the optional and variable features in the decision model).
- **Production mechanism and process.** The means for composing and configuring products from the software asset inputs. Product decisions are used during production to determine which software asset inputs to use and how to configure the variation points within those assets.
- **Software product outputs.** The collection of all products that can be produced for the product line. The scope of the product line is determined by the set of software product outputs that can be produced from the software assets and decision model.

Some of these ideas have it roots in approaches such as Program Factoring [127] or Domain Engineering [128]. The characteristic that distinguishes software product

lines from previous efforts is predictive versus opportunistic software reuse. Rather than put general software components into a library in hopes that opportunities for reuse will arise, software product lines only call for software artifacts to be created when reuse is predicted in one or more products in a well defined product line. To this end, the main objectives of software product lines are as follows.

- **Capitalize on commonality** through consolidation and sharing within the software asset inputs, thereby avoiding duplication and divergence.
- **Manage variation** to reduce the time, effort, cost and complexity of creating and maintaining a product line of similar software systems. This is achieved by clearly defining the variation points and decision model, thereby making the location and dependencies for variation explicit.

That is, Software Product Line Engineering (SPLE) optimizes the development of individual systems within an application domain by leveraging their common characteristics and managing their differences in a systematic way. In SPLE, individual systems can be built rapidly from reusable assets, such as a set of components and/or a common platform.

5.2.2 Model Driven Software Product Lines

Generative software development [129] and related approaches, such as Software Factories [130], have been propagating the integration of software product lines and model-driven software development; also, entire workshops have been recently dedicated to this topic [131, 132].

Model-Driven Development (MDD) aims at capturing every important aspect of a software system through appropriate models. Compared to implementation code, models capture the intentions of the stakeholders more directly, are free from accidental implementation details, and are more amenable to analysis. In MDD, models are not just auxiliary documentation artifacts; rather, they are source artifacts and can be used for automated analysis and/or code generation.

SPL engineering and MDD are not only complementary, but their integration bears the potential for significant synergies. While MDD can help us represent

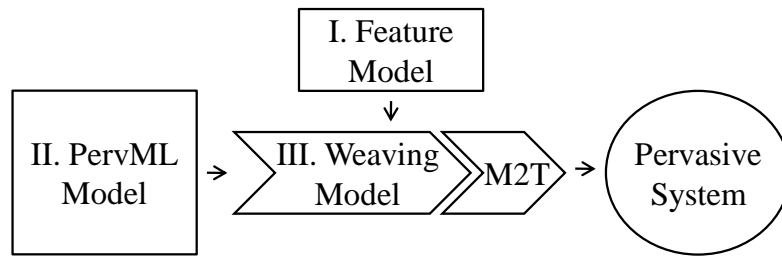


Figure 5.3: SPL following the MDD Approach

different aspects of a product line more abstractly, SPL engineering provides a well-defined application scope, which puts the development and selection of appropriate modelling languages on a sound basis. Furthermore, the automated analysis and code generation afforded by precise models can help us to automate the creation of system configurations. MDD provides effective techniques for conveying the results of specifying variability as follows:

- Metamodelling, which defines type systems that precisely express key abstract syntax characteristics and static semantic constraints associated with product-lines for particular application domains, such as pervasive systems, mobile computing or resilience systems.
- Domain-specific languages (DSLs), which provide notations that are guided by and extend metamodels to formalize the process of specifying product-line structure, behavior, and requirements in a domain.
- Model transformations and code generators that ensure the consistency of product-line implementations with analysis information associated with functional and QoS requirements captured by structural and behavioral models.

Key advantages of using MDD in conjunction with variability of SPLs are (1) rigorously capturing the commonalities and variabilities in a family of systems and (2) helping automate repetitive tasks that must be accomplished for each product instance.

Figure 5.3 shows how to combine modelling and model transformations to develop an MDD-SPL for Smart Homes. First, the assets of the MDD-SPL are model

elements which describe a family of Smart Homes. These model elements conform to the metamodel of PervML which is a DSL for Smart Homes. Second, the decision model is another model which specifies the aspects or characteristics (named features) of a particular Smart Home. Third, a weaving model projects the features on the DSL for the purpose of scoping the domain model. Finally, the output system is obtained through a model (scoped DSL) to text (Java code) transformation.

Given such MDD-SPL, we argue that the modelling effort made to define the MDD-SPL is not only useful for producing the system but also for providing autonomic behaviour during execution. The knowledge previously captured in variability models can be used to describe the variants in which a system can evolve. Furthermore, variability models can assist the execution to determine the steps that are necessary to reconfigure a software system. Next, we describe the models which conform our MDD-SPL (see Figure 5.3), and how these models can enable a system to achieve autonomic behaviour.

I. Feature Modelling

Feature Modelling is 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 through variability relationships such as optional, mandatory, single-choice and multiple-choice, and are optionally connected by cross-tree constraints such as requires or excludes.

There are many proposals about the type of the relationships and the graphical representation of feature models [133]. We have chosen the MOSKitt Feature Model as the modelling language because it supports feature reasoning (by means of the FAMA framework [21]) and also because it has good tool support¹.

The Feature Model of Figure 5.4 describes a Smart Home with *Automated Illumination*, *Blind Control* and *Security*. The grey features represent the features of the smart home, while the white features represent potential variants since they may be activated in the future. For instance, the smart home initially provides automated

¹<http://www.pros.upv.es/labs/projects/mfm>

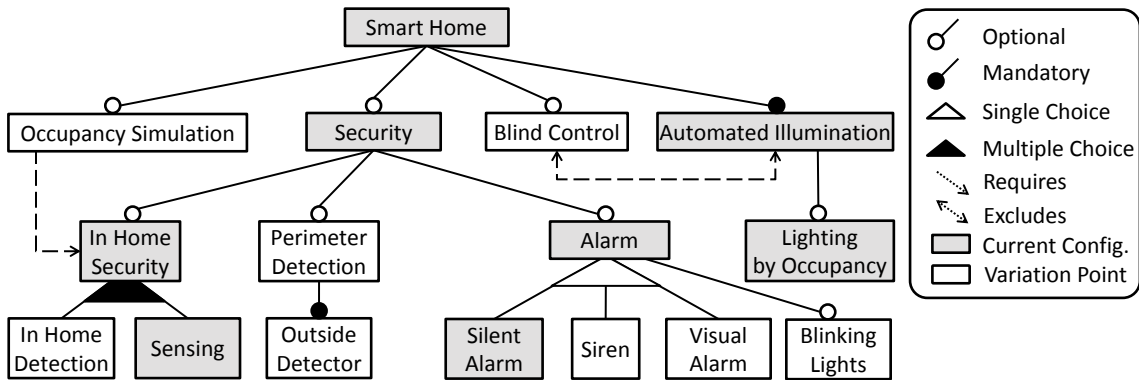


Figure 5.4: Feature model of a smart home.

lighting and a security system. This security system relies on in-home detection (inside the home) and a silent alarm. The system can potentially be upgraded with perimeter presence detection and other alarm to enhance home security.

Let $\llbracket FM \rrbracket$ denote the set of all Features (active or inactive) in a Feature Model; we define the Current Configuration (CC) of a system as the set of all active features (F) in its Feature model.

$$CC \stackrel{\text{def}}{=} \{F\} \mid F \in \llbracket FM \rrbracket \wedge F.state = Active \wedge CC \subseteq FM$$

For example, the CC of the Feature Model in Figure 5.4 is expressed as follows:

$$CC_{Figure5.4} = \{SmartHome, Security, InHomeSecurity, Sensing, Alarm, AutomatedIllumination, SilentAlarm, LightingByOccupancy\}$$

II. Pervasive System Modelling

Pervasive Modelling Language² (PevML) [24] is a Domain Specific Language (DSL) for describing pervasive systems using high-level abstraction concepts. This language is focused 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, and services can also react to

²<http://www.pros.upv.es/labs/projects/pervml>

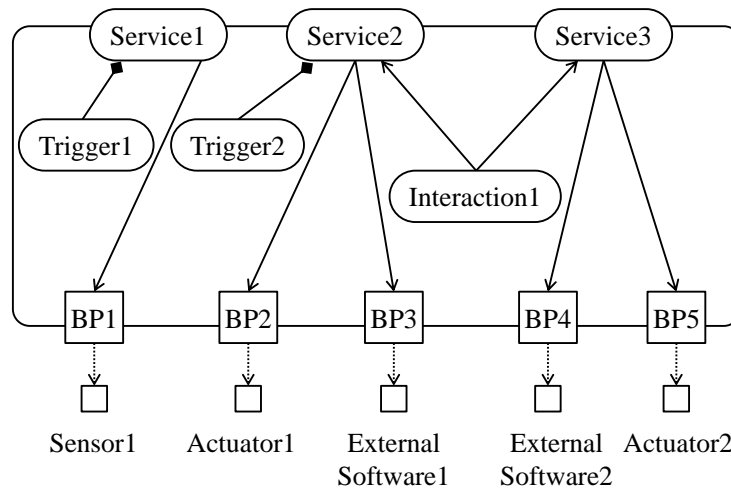


Figure 5.5: Main concepts of PervML Pervasive Systems

changes in the environment. This DSL have been successfully applied to develop solutions in the smart home domain [134].

The main concepts of the PervML language are: (1) a *Service* coordinates the interaction between devices to accomplish specific tasks (these devices can be hardware or software entities); (2) a *Binding provider* is a device 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 devices. Figure 5.5 illustrates the relationships between these concepts.

The PervML language provides different models to specify the services (*Service and Interaction Models*) and devices (*Binding Provider and Functional Models*) of a pervasive system. See [135] for a detailed description of PervML models. To address variability in pervasive systems, we focus on the *Structural Model* of PervML. The *Structural Model* specifies (1) the components that conform a particular configuration of the system (services and devices), and (2) how these components are connected among them.

Figure 5.6 shows two Smart Home configurations according to the concrete syntax of the *Structural Model*. Services are represented by a circle, devices are represented by a square, and the connections among services and devices are represented by lines.

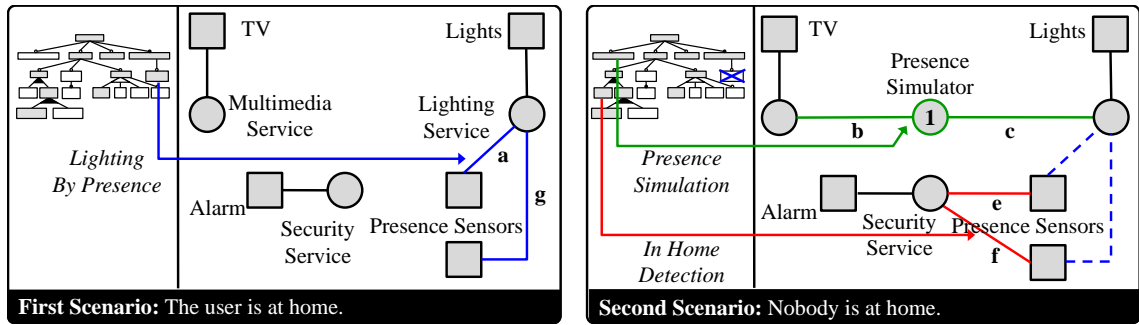


Figure 5.6: Two configurations of a Pervasive System.

Left of Figure 5.6 shows a *User at Home* configuration (which is the configuration of Figure 5.4), while right of Figure 5.6 shows a *Nobody at Home* configuration. Comparing both configurations, movement sensors are no used for lighting (left); they are used for providing information to the security service instead (right). In addition, the Occupancy simulation service is activated at *Nobody at Home* configuration, and the connections that are required for this service to communicate with multimedia, lighting and security services are established.

III. Weaving Model: Tracing Features to PervML elements

Although a feature model can represent commonalities and variabilities in a very concise taxonomic form, features in a feature model are merely symbols. Mapping features to other models, such as behavioral or data specifications, gives them semantics. Next, we show how to perform this mapping by means of a weaving model [136]. This weaving approach enables us for scoping and configuring PervML models from a set of given Features.

Weaving models are used to define and to capture relationships between models elements. Relationships between model elements are present in many different application scenarios, such as specification of transformations, traceability, or model alignment. We use the weaving models to define the relationships between Features and model elements of a DSL.

Consider a weaving model (Mw) between a Feature model (Fm) and a PervML model (Pm), denoted by the triple $\langle Mw, Fm, Pm \rangle$. Mw contains a set of elements that link a set of elements of Fm with a set of elements of Pm. The elements of a

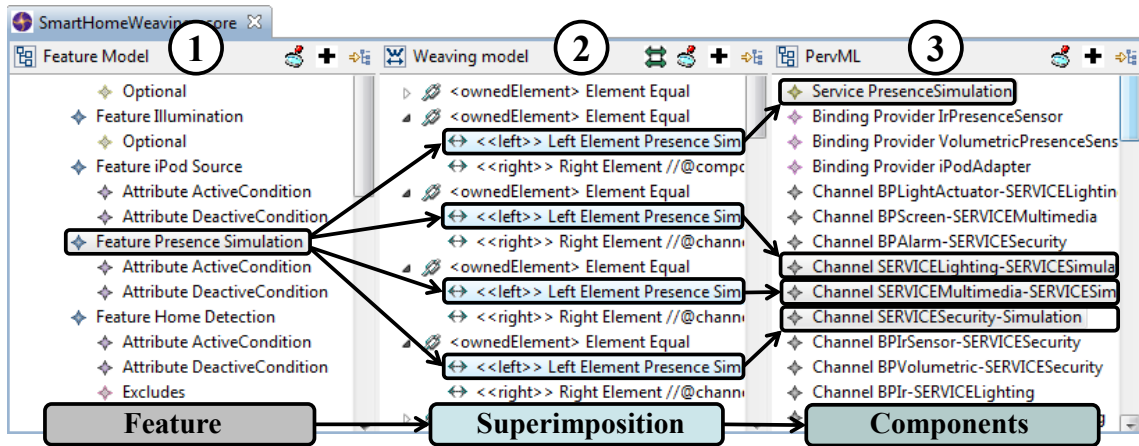


Figure 5.7: Snapshot of a Feature Model.

Mw to support link management using weaving models are as follows.

- **WModel** represents the root element that contains all model elements. It is composed by the weaving elements and the references to woven models.
- **WLink** express a link between model elements that has simple linking semantics. Its semantic has to be refined according to the use of the weaving model.
- **WLinkEnd** (also know as link endpoint) represents a linked model element. This element enables the creation of N-ary links.

Since we are specifying links between Features and PervML elements, a Fm contains Features from the $[[FM]]$ set (active or inactive features), and Pm contains elements of PervML such as: services, devices or connections among the formers. In our case, a WLink \in Mw indicates that a given element $p1 \in Pm$ will be included in the resulting PervML configuration if and only if $f1 \in Fm$ is active. That is, by means of the weaving model, the PervML configurations is instantiated through the activation/inactivation of features in the Feature Model.

Finally, the weaving model Mw does not contain the concrete elements $p1$, or $f1$, but a reference that enables to access them in the containing models (Pm and Fm). This is the minimum information to have a way to access and to uniquely identify each linked element.

Figure 5.7 shows a weaving model between the Feature model and the PervML model of a Smart Home using ATLAS Model Weaving tool [137]. The Structure model of PervML (column 3, Figure 5.7) specifies the services, devices and connections among them. The Smart Home features are specified by MOSKitt Feature Model (column 1) and Atlas Model weaving (column 2) establishes the relations between features and architecture components.

In order to query a weaving model to identify which PervML elements support a certain feature, the **Superimposition** operator (\odot) is defined. The Superimposition takes a Feature and returns the set of components and channels related to this Feature. Some examples of the relationship between Features and the Smart Home of Figure 8.2 are as follows:

$$\odot(\textit{LightingByOccupancy}) = \{a, g\}$$

$$\odot(\textit{OccupancySimulation}) = \{1, b, c, d\}$$

$$\odot(\textit{InHomeDetection}) = \{e, f\}$$

For example, *LightingByOccupancy* is supported by the connections labeled as *a* and *g* in Figure 5.6. As well as, *Occupancy Simulation* is supported by the connections *b*, *c*, *d* and the service *1*.

5.3 Specifying Reconfigurations through Feature Models

In the context of Software Product Line engineering, the focus of variability models has been on the efficient derivation of customized product variants that, once created, keep their properties throughout their lifetime. That is, although Software Product Line engineering recognizes that variation points are bound at different stages of development, and possibly also at run-time, it typically binds variation points before delivery of the software.

We believe that variability models can be also used to provide a richer semantic base for run-time decision-making related to system adaptation. For example,

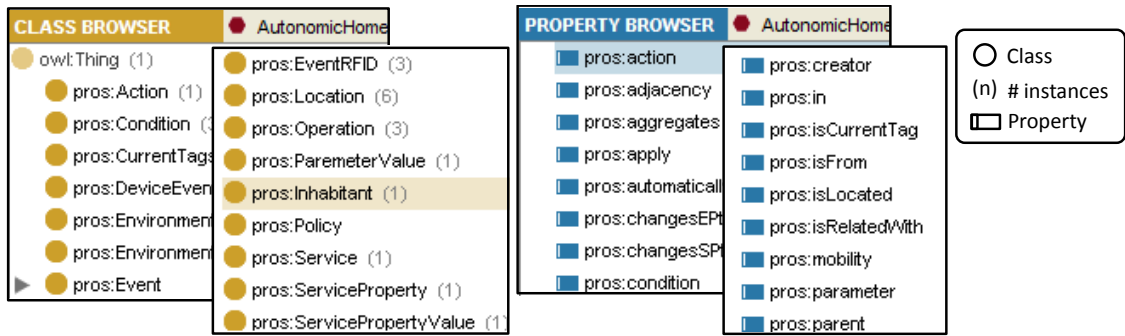


Figure 5.8: OWL Ontology for Autonomic Homes.

variability models can be used to determine how a system should move from a consistent architecture to another consistent architecture. This model-based management of executing systems can play a significant role as we move towards implementing the key self-* properties associated with autonomic computing. Specifically, we argue that a system can activate/deactivate its own features dynamically at run-time according to the fulfilment of Context Conditions.

For context modelling, we use an ontology-based context model that leverages Semantic Web technology and OWL (Web Ontology Language) [122]. OWL is an ontology markup language that enables context sharing and context reasoning. In the artificial intelligence literature, an ontology is a formal, explicit description of concepts in a particular domain of discourse. It provides a vocabulary for representing domain knowledge and for describing specific situations in a domain. An ontology-based approach for context modelling lets us describe contexts semantically and share common understanding of the structure of contexts among users, devices, and services. The main benefit of this model is that it enables a formal analysis of the domain knowledge, such as performing context reasoning using first order logic.

An ontology represents our context model structure. The ontology is described in OWL as a collection of RDF triples, in which each statement is in the form of (subject, predicate, object). The subject and object are the ontology objects or individuals and the predicate is a property relation defined by the ontology. For instance, (John, Location, garden) means that John is located in the garden. Figure 5.8 shows our current ontology for context modelling in Autonomic Homes. For more information about the structure and population of this ontology see [138].

Given such ontology, we define context conditions as boolean expressions that check for values in the Smart Home ontology. Examples of these contextual conditions are:

```

1 NewVolumetricSensor = (Volu360, state, "installed") &&
2                       (Volu360, location, "In-Home")
3     AlarmFailure = (CentralAlarm, state, "failure")
4     EmptyHome = (MovSensorMainDoor, movement, false) &&
5                 (MovSensorStairs, movement, false)

```

The first condition (`NewVolumetricSensor`) detects that a movement sensor is being connected in a specific location. The system can activate the appropriate features to integrate the new device in order to offer *self-configuration*. The second condition (`AlarmFailure`) detects the situation where the alarm is not working, so an alternative mechanism can be used instead of the alarm (*self-healing*). Finally, the third condition (`EmptyHome`) is fulfilled when none of the movement detection sensors is perceiving movement. This can be used to trigger the activation of both the *In Home Detection* and the *Occupancy Simulation* features when home members leave home, to achieve *self-adaptation*.

Since a given condition can trigger the activation/deactivation of several features, we define the **Resolution** concept (**R**) to represent the set of changes triggered by a condition. A *resolution* is a list of pairs where each pair is conformed by a Feature (**F**) and the state of the feature (**S**). Each *resolution* is associated to a context condition and represents the change (in terms of feature activation/deactivation) produced in the system when the condition is fulfilled.

$$R \stackrel{\text{def}}{=} \{(F, S)\} \mid F \in \llbracket FM \rrbracket \wedge S \in \{Active, Inactive\}$$

For instance, the condition `EmptyHome` is associated to the following resolution:

$$R_{EmptyHome} = \{(OccupancySimulation, Active), \\ (InHomeDetection, Active), \\ (LightingByOccupancy, Inactive)\}$$

This means that when the Smart Home senses that it is empty (according to the condition), it must reconfigure itself to deactivate *LightingByOccupancy* and to activate both *OccupancySimulation* and *In-HomeDetection*. The Feature Model at run-time enables the Smart Home to perform this reconfiguration.

Our approach argues the use of the above Feature Model as the knowledge to drive the autonomic behaviour of the system. Specifically, we argue that a system can query the above models in order to bind its own variation points, initially when software is launched to adapt to the current context, as well as during operation to adapt to changes in the context.

5.3.1 Evaluation of the Autonomic Behaviour through Feature Models.

To determine the level of autonomic behaviour that can be achieved by means of Feature Model reconfigurations, we make use of a state machine since, in practice, engineers use state machines to represent and check adaptation policies [112]. Figure 7.9 illustrates how a simple feature model, which consists of four features only (see a), defines eight possible system configurations $C1$ to $C8$ (see c). When a designer defines a condition for the activation of a system feature by means of a resolution $\neg R_{ConditionX}$, $R_{ConditionY}$, or $R_{ConditionZ}$ (see b), he/she is expressing the transitions between different system states (see d) in a declarative manner, without the need for an exhaustive definition of each state transition or the transitions derived from the composition of states. In the example, a single resolution such as $R_{ConditionY}$ results in eight transitions among system variants (represented as dashed-arrow lines in the figure).

The presented parallelism shows how a Feature Model hides much of the complexity in the definition of the adaptation space for an autonomic system. In the example of Figure 5.4, the feature model containing 18 features represents more than 200,000 states, and the three resolutions for this feature model (*NewVolumetricSensor*, *AlarmFailure* and *EmptyHome*) define more than 600,000 transitions among system variants. Feature Models provide an intensional description of the possible states of the system, as opposed to extensionally describing each possible state.

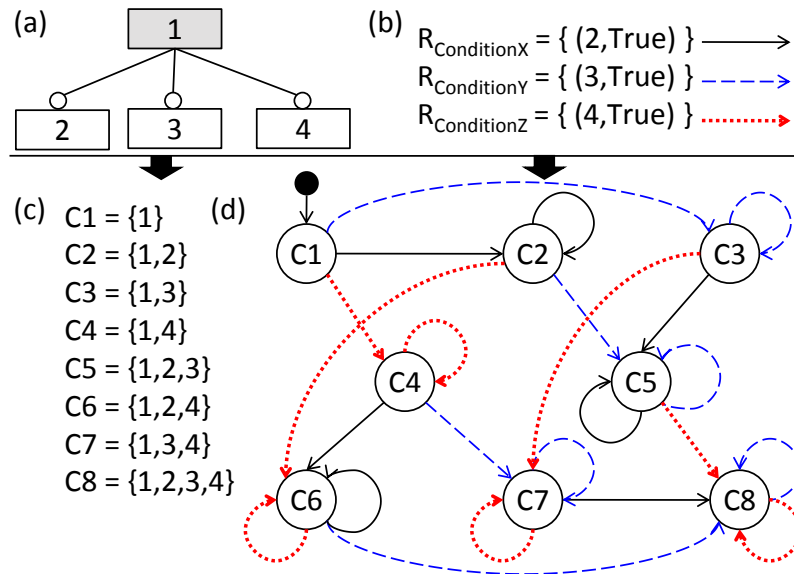


Figure 5.9: Visualizing variability as an adaptation space.

5.4 Model-Based Validation of Reconfigurations

For dependable systems, it is indispensable to have a means to analyze the reconfigurations before performing them. However, the simultaneous fulfillment of Context Conditions leads to an exponential growth in the number of possible system reconfigurations. This presents a major problem, since reasoning on a huge number of reconfigurations becomes too time consuming at run-time when considering the response time in the Smart Home domain that we are addressing [139]. Furthermore, these activities are very complex and error prone and hence pose the need for a sound and seamless engineering support.

To address the above problem, our approach validates the configurations resulting from the simultaneous fulfillment of Resolutions at design time. Therefore, unexpected configurations such as enabling in-home intrusion detection systems can be avoided when inhabitants are at home. In particular, we analyse Feature Models by means of the Feature Model Analyser Framework [21] (FAMA). This framework implements the automated analysis of Feature Models using Constraint Satisfaction Problems [110]. In particular, we focus on two operations: Check and Filter [140].

- The **Check operation** takes a configuration as input and it returns a value

that determines whether or not the configuration is valid according to a Feature Model. For instance, the $R_{\text{Invalid Config}}$ described below is not valid according to the Feature Model of Figure 5.4 because *OutsideDetector* must already be activated in order to activate *PerimeterDetection*.

$$R_{\text{Invalid Config}} = \{(PerimeterDetection, Active), (OutsideDetector, Inactive)\}$$

- The **Filter operation** acts as a limitation for potential configurations. This operation enables designers to identify the configuration that will arise from the simultaneous fulfillment of several Resolutions. In the Smart Home example, the simultaneous fulfillment of the conditions *EmptyHome* and *Comfort* generates the following resolution:

$$R_{\text{Empty \& Comfort}} = \{(OccupancySimulation, Active), (InHomeSecurity, Active), \\ (LightingByPresence, Inactive), (BlindControl, Active), \\ (AutomatedIllumination, Active)\}$$

We can also apply the check operation to $R_{\text{EmptyHome \& Comfort}}$. If both conditions are true in the Smart Home context, then the check operation determines that the resulting configuration is invalid. The configuration is invalid because *AutomatedIllumination* and *BlindControl* cannot be in a configuration at the same time (see Figure 5.4).

The combination of these two operations turns out to be a powerful mechanism to identify potential problems at design time. Actually, even though variability decisions are taken in multiple stages (resolutions) to form a complete configuration iteratively, we can still identify invalid configurations.

However, it is interesting not only obtaining a valid-invalid conclusion but knowing the reasons why that conclusion is inferred. For example, if we find an error such as the given by $R_{\text{EmptyHome \& Comfort}}$, then we may be interested in the relationships that make this error appearing. So we can use this information to assist on error repairing. This transverse operation is commonly known in Feature model analysis community as explanation [141, 109] and may be used in conjunction with any deductive operation such as the Check operation.

Explanation operations intend to extract relevant information from feature models to assist on decision making. Specifically, this operations allow to obtain conjectures about why things happen. FAMA framework provides a catalog of explanation operations on Feature models [142]. For each deductive operation (i.e., Check operation), FAMA propose “why?” and “why not?” questions. “Why?” questions are asked when a deductive operation has a solution. “Why not?” questions intend to find an answer for a deductive operation that has no solution.

The Check operation may obtain a negative response when inconsistencies are found. Whenever the Check operation detects an invalid configuration. It is necessary to obtain further information about the relationships that are making the product impossible to derive. In the example of $R_{\text{EmptyHome \& Comfort}}$, the Check operation identifies that the resulting configuration is invalid. The explanation operation (“Why is a configuration not valid”) explains this unexpected result by detecting the excludes constrain between *AutomatedIllumination* and *PresenceSimulation* as the relationships that are causing it.

The objective of Explanation operations is to find the reasons that explain the inconsistent situation. The results of this explanation operations help system designer to refine both Feature model constraints and Resolutions. Hence, existing techniques for variability analysis can be used as an step in obtaining safe systems free of unsafe reconfigurations.

5.5 Applying the approach to other Variability Language: Autonomic Behaviour through Common Variability Language

The use of variability models for enabling autonomic behaviour is the central idea of this work. Although feature models have been used for capturing variability, our approach can be applied to other Variability modelling languages.

Variability models for system families come in two very different forms: as pure feature models that are independent of any design or implementation model, or as

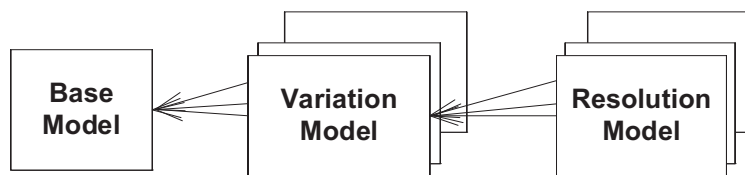


Figure 5.10: Base-Variation-Resolution Approach.

variability models that are related to a base model: elements of the base model will become elements of specific models (or not) according to resolutions of related variability models.

In the above section, feature models were applied due to the availability of tool support for analysis, but other modelling approaches may be more suitable for certain types of tasks. In this section, we introduce the key steps to apply our technique to another representative modelling language: the Common Variability Language (CVL) [121]. We have chosen CVL because it pursues OMG standardization of variability modelling and management.

CVL is based in the Base-Variation- Resolution approach (BVR-approach) which argues to define orthogonal variability models that apply to a single base model (see Figure 5.10). The BVR approach described in [143] was developed within the Families project [144]. The main focus in that work was variability models for base models in general purpose modelling languages like UML, so some of the variability mechanisms in the language for making variability models relied on the existence of certain base model language mechanisms. CVL reports on work within the ITEA project MoSiS (ITEA 2 - ip06035) to apply (and thereby further elaborate) the Families variability language also to DSLs that may not have the language mechanisms of general purpose languages. The aim is to come up with a variability language that may enhance both DSLs and general purpose languages.

The motivation of CVL is to separate variability modelling from the base domain modelling. CVL is suitable for modelling variability of models in any base language such as DSLs or UML. The CVL approach leaves the base domain modelling to the DSL while the variability is treated with CVL (see Figure 5.11). This separation between the DSL and the variability language provides a good separation of concerns

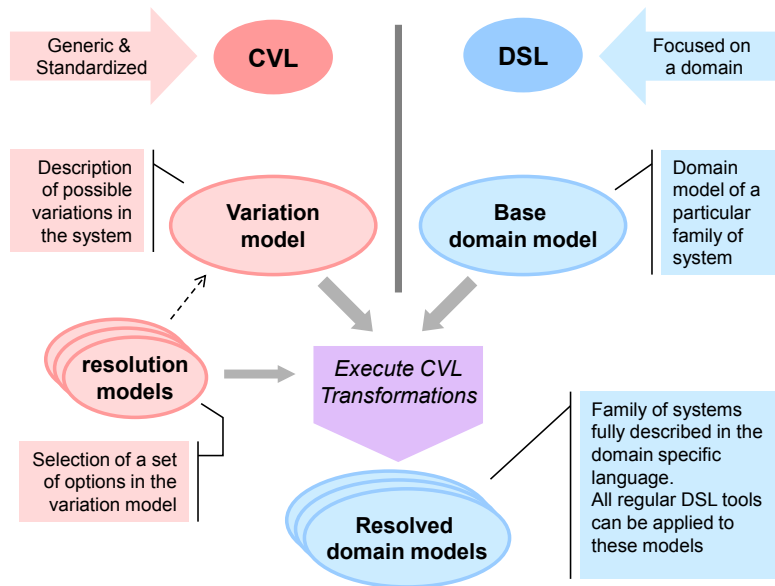


Figure 5.11: Modelling Variability with CVL.

and of developing efforts. The domain experts can concentrate almost exclusively on the DSL.

As illustrated in Figure 5.12, a CVL specification consists of one variation-model that is applied to one base-model (DSL or UML), and several resolution-models. The variation-model defines a set of alternatives for model fragments in the base-model. A fragment can be any arbitrary part of the base model, including a set of elements and their relationships. Finally, a resolution model specifies which fragments of the base-model are replaced by fragments of the variability-model.

We have successfully applied our modelling approach using CVL [126]. First, we built a base model using PervML (see left of Figure 5.12). Then, we specified valid replacement fragments for this base-model: Simulation and Sensing. Finally, we defined resolution models that specify substitutions between placements and replacement in order to synthesize a new PervML configuration. For instance, to achieve a *nobody at home* configuration (see right side of Figure 8.2), the following fragment substitutions can be applied:

$$R_{EmptyHome-CVL} = \{(Simulation, Basic), (Sensing, Security)\}$$

The $R_{EmptyHome-CVL}$ is performed as follows. Initially, an empty fragment is connected to a multimedia service and a lighting service. This empty fragment is

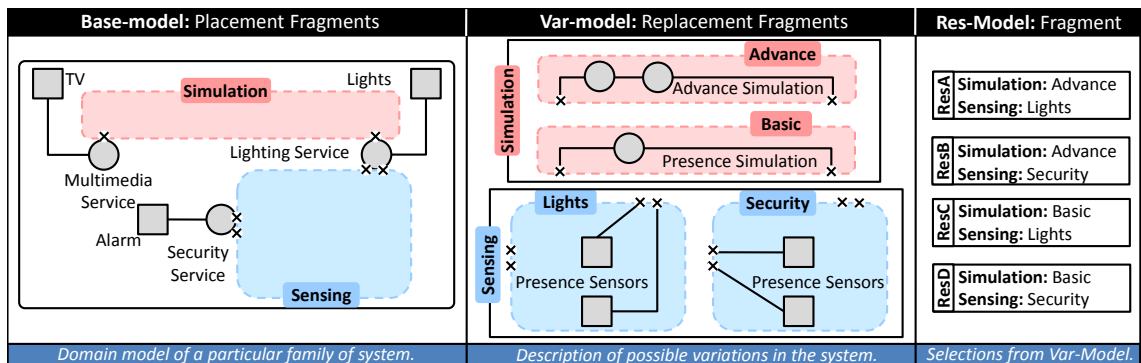


Figure 5.12: Applying CVL for Autonomic Homes.

replaced with a fragment consisting of three model elements (presence simulation service and two communication channels). Then, the empty fragment of Sensing is replaced by the Security fragment. These two fragment substitutions lead to the *nobody at home* configuration depicted on the right side of Figure 8.2.

The reconfiguration process based on model substitutions is supported by the CVL tools, which implement this variability transformation. This transformation generates a new and a modified base model, which contains the elements representing the substitutions made.

Figure 5.13 shows both Feature modelling and CVL side by side. On the one hand, Feature modelling specifies the whole system family by means of the DSL model. Then, this family is scoped through the superimposition of features. On the other hand, CVL specifies placements in a base DSL model. For each placement, CVL also specifies a set of possible fragments. Then, the DSL is configured by setting a particular fragment for each placement.

The advantage with the feature-based approach is that model elements subject to variability are clearly marked, while the disadvantage is that base models are cluttered with variability specifications. The advantage with the CVL approach is that there may be more than one variability model for each base model, which contributes to manage complex variability specifications. However, the main advantage disadvantage is that base model elements subject to variability are not clearly marked.

Based on our variability modelling experience, expressing the whole family of sys-

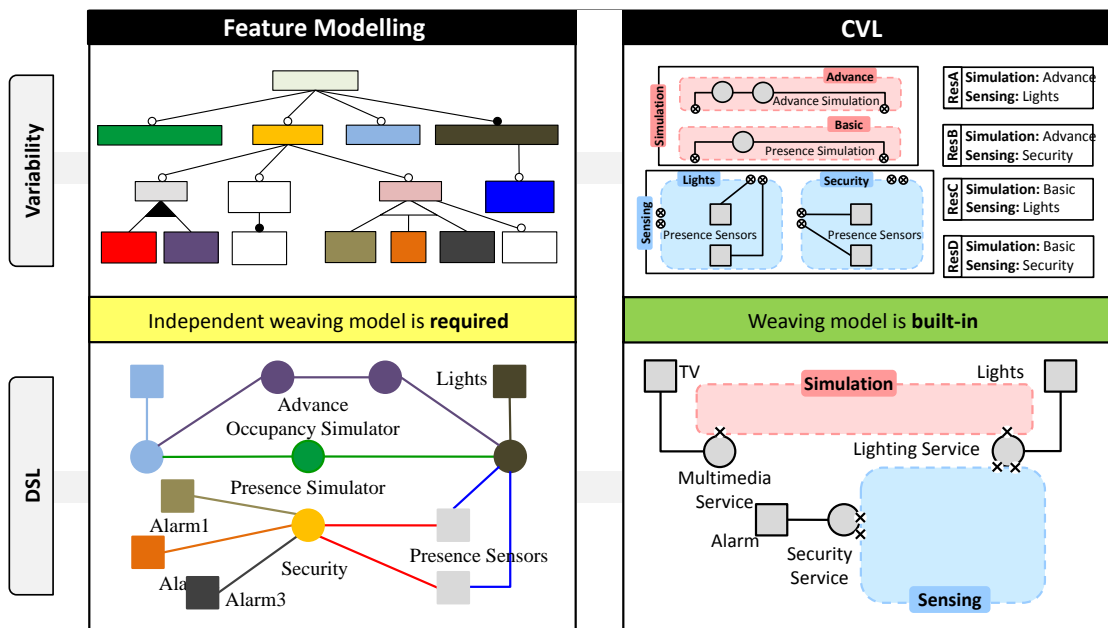


Figure 5.13: Feature Modelling and CVL.

tem by means of a DSL model (as feature modelling promotes) leads to often fairly complex models. Whereas, expressing variability by fragments of models leads to compact and simple DSL models. We believe that the fragment-based approach manages variability in a more structured manner, and it also simplifies the specification of new variants as the system family evolves.

5.6 Conclusions

In this chapter, we have argued that the modelling effort made at an MDD-SPL is not only useful for producing the system but also for providing autonomic behaviour during execution. The knowledge previously captured in variability models is used to describe the variants in which a system can evolve. We have also obtained theoretical results about the autonomic behaviour specified by Variability Models.

Since the models that form the basis for reconfiguration are available at design time, we have shown how to validate configurations in an early stage of the development process without first implementing them. Furthermore, we have automated this step combining analysis operations of the FAMA framework. Besides, by means of Explanation operations, we are able to get the reasons that explain inconsistent

situations.

We have also validated that our work can be applied to different modelling languages. Specifically, we have applied our approach using both Feature Models and the Common Variability Language. On the one hand, Feature models were chosen due to the availability of tool support for analysis. On other hand, CVL was chosen because it pursues OMG standardization of variability modelling and management.

We believe that the use of variability models at run-time brings new opportunities for achieving autonomic capabilities. Variability models provide a richer semantic base for run-time decision-making related to system adaptation. This is done by means of a planned reutilization of the efforts invested at design time.

Next chapter shows how Variability Models at run-time guide the choice of system variants in response to context changes. Furthermore, we also show how the model operations to query and update variability models for run-time reconfiguration. These operations indicate how system components should be reorganized for the reconfiguration in order to move from one configuration of the system to another configuration.

Chapter 6. ACHIEVING AUTONOMIC COMPUTING THROUGH MODELS AT RUN-TIME

“Freedom to be your best means nothing unless you’re willing to do your best.”

– Colin Powell (1937-nowadays).

6.1 Overview of the Chapter

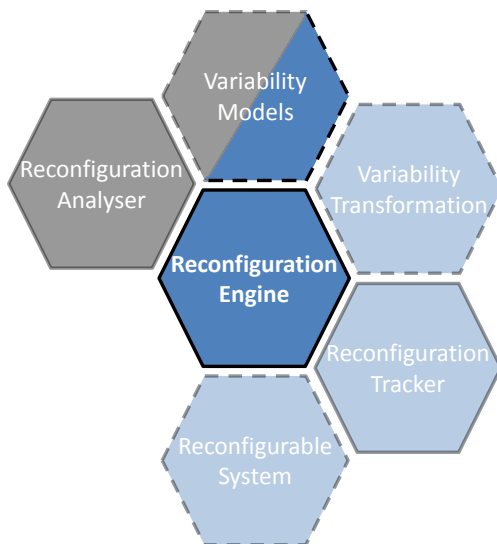


Figure 6.1: Scope of Chapter 6

To achieve Autonomic Computing, our work makes use of models at run-time [145, 146] (Variability Models and Ontologies) and Reconfigurable Architectures [18]. Run-time models specify the possible configurations of a Smart Home, while a Reconfigurable Architecture can be rapidly retargeted to a specific configuration. That is, this work uses modelling techniques to define the bounds in which a system can evolve by using at run-time the variability models that are available at design time.

First, this chapter presents how a system architecture can represent a family of software systems where the configuration can be updated while the system is operational. Specifically, it defines how a set of components cooperate to change from one configuration of the system family

to another. The realization of this Reconfigurable System Architecture has been performed by means of the OSGi Framework

Second, this chapter presents our approach based on Variability Models and Ontologies to drive the reconfiguration of the system architecture in response to changes in the environment. In response to these changes, the system itself can query these variability models in order to determine the necessary modifications to its architecture. The approach is presented in the context of a Smart Home whose services are dynamically reconfigured.

Third, to support the proposal, a Model-based Reconfiguration Engine (named MoRE) was developed. MoRE implements (1) the operations that are in charge of determining how the system should evolve and (2) the actions for modifying the system architecture accordingly. Thus, MoRE enabled systems make use of the knowledge captured by variability models as if they were the policies that drive the autonomic evolution of the system at run-time.

Finally, in our proposal, models are leveraged at run-time as is, without modification. That is, we keep the same model representation at run-time that we use at design-time: the XML Metadata Interchange (XMI) standard. This is a novel feature in the context of variability models at run-time since it avoids the definition of technological bridges. Therefore, the same technologies used at design-time for manipulating XMI models can be applied at run-time. Our experimentation shows the feasibility of this approach from the point of view of efficiency.

6.2 Renconfigurable System Architectures

A software architecture can represent a family of software systems characterized by the similarities and variations among the members of the system family. Software configuration is the process of adapting the architecture of the system family to create an architecture for a specific system family member. Many approaches of system families address the configuration of a family member prior to system operation. A more difficult problem is how to change the configuration of a product family member after it has started to operate. Reconfigurable architectures can address this

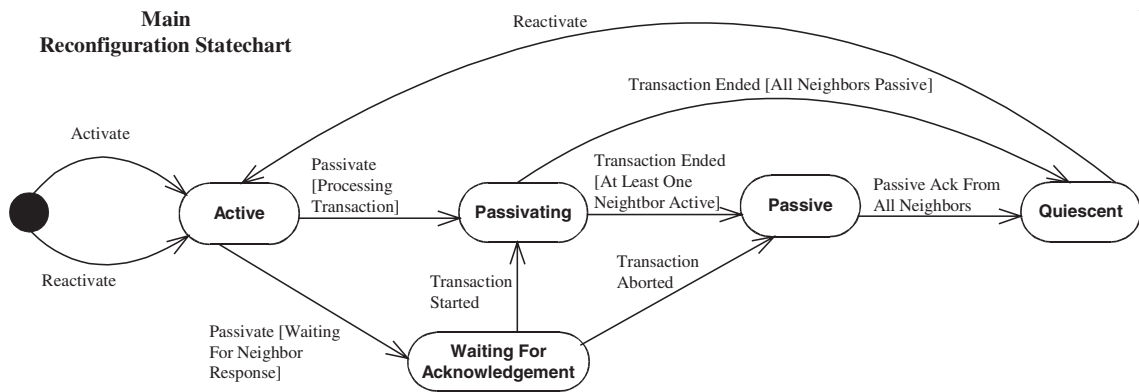


Figure 6.2: Reconfiguration Pattern of a DSPL Architecture.

problem by changing the configuration of the running system at run-time.

Reconfigurable architectures promotes that each architecture component is designed to be capable of transitioning to a state where it can be reconfigured. To achieve this behaviour, we apply software reconfiguration patterns [147] for dynamic reconfiguration in system families. Software reconfiguration patterns provide a solution to a reconfiguration problem where the configuration needs to be updated while the system is operational.

Specifically, we use the *Decentralized Control System Reconfiguration* Pattern [148], which is widely used in distributed control systems such as Smart Homes [149]. In this pattern, components notify each other if going to a passive state. Notified components can cease the communicate with its neighbor component (which is going to a passive state), but can continue with other component communications.

Figure 6.2 shows the reconfiguration pattern by means of a Reconfiguration Statechart. A reconfiguration statechart defines the sequence of states that a component goes through during reconfiguration. In a Reconfiguration Statechart a component is either in the Active, Passive, Quiescent, Passivating or Waiting state .

- **Active State.** An operational component is in the Active state.
- **Passive State.** A component is in the Passive state when it is not currently engaged in a transaction, and it will not initiate new transactions.
- **Quiescent State.** A component transitions to the Quiescent state if it (1) is Passive, (2) is not currently engaged in servicing a transaction, and (3) no

transactions have been or will be initiated by other components which require service from this component.

- **Passivating State.** A component is in the Passivating state when it is disengaging itself from any transactions that (1) it has been participating and (2) it has been initiating.
- **Waiting State.** A component is in the Waiting For Acknowledgement state if it has sent notification message(s) to interconnected components to inform them of its need to go passive, and then it is waiting for positive acknowledgements.

In this reconfiguration pattern (see Figure 6.2), if a passivate command arrives while the component is processing a transaction, the component transitions to Passivating state. When the transaction ends, the component either transitions to Passive state, because one of its neighbors is still active, or it transitions directly to Quiescent state if all neighbors are passive. If the component receives a passivate command after it has requested its neighbor to start a transaction but before it has received a response (i.e., while the condition *Waiting For Neighbor Response* is true), then it sends a notification to its neighbor component informing it that it wishes to cancel the outstanding request. Then it transitions to the *Waiting For Acknowledgement* state. If the neighbor responds that the transaction has started, the component transitions to the Passivating state. Otherwise, if the neighbor responds that the transaction was aborted, the component transitions to the Passive state. The reason for the *Waiting for Acknowledgement* state is to prevent a race condition.

This *Decentralized Control System Reconfiguration* pattern enables a system architecture to be reconfigured at run-time after it has been deployed. Furthermore, this pattern provides the following properties to the resulting architecture: (1) Non interference with those parts of the application that are not impacted by the reconfiguration, and (2) during reconfiguration, impacted components must complete their current computational activity before they can be reconfigured.

6.2.1 The OSGi Framework: A Realization of the Renconfigurable System Architecture

The Open Services Gateway Initiative (OSGi) framework [124] provides general-purpose, support for deploying extensible Java-based service applications known as bundles. An OSGi service platform is an instantiation of a Java virtual machine, an OSGi framework, and a set of bundles.

Running on top of a Java virtual machine, the framework provides a shared execution environment that installs, updates, and uninstalls bundles without needing to restart the entire system. Bundles can collaborate by providing other bundles with application components called services. An installed bundle might register zero or more services with the framework's service registry. This registration advertises the services and makes them discoverable through the registry so that other bundles can use them. The framework also manages dependencies among bundles and services to facilitate coordination among them.

It is possible to deploy a new bundle in an OSGi service platform to provide application functions to other bundles. A bundle can register services with the framework service registry. In this case, the service implementation (that is, the service object), which is represented by its service interface, is what actually gets registered.

Bundles can discover services offered by each other by querying the service registry using a simple service discovery interface. When a bundle queries the registry, it obtains references to actual service objects registered under the desired service interface name.

The framework manages dependency among bundles that offer and use a given service. For example, when a bundle is stopped, the framework automatically unregisters all services that the bundle registered. Also, service events can notify a bundle when a service from other bundles is registered, modified, or unregistered.

The OSGi capabilities to install, start, restart and uninstall components without having to restart the entire system enabled us to implement the *Decentralized Control System Reconfiguration* Pattern. This pattern describes how a component needs to

transit from an active (operational state) to a quiescent (idle) state in order to perform the system adaptation. All those components that are not relevant for the current configuration are in a catalog of quiescent components. These quiescent components do not consume processor or memory resources, but they are ready to be started at any time.

Once a component transits to an active state, the *Decentralized Control System Reconfiguration* Pattern specifies that the component has to establish communication with other components. These communication channels (also called bindings) are implemented using the OSGi Wire Class. An OSGi Wire is an enhanced implementation of the publish-subscribe pattern that is oriented to dynamic systems. In particular, an OSGi Wire implements the whiteboard pattern. The whiteboard pattern has event listeners that register themselves as a service within the OSGi framework. When the event source has an event object to deliver, the event source calls all event listeners in the service registry.

The Wire Admin Service in OSGi service platform addresses the intercomponent eventing mechanism introduced by the reconfiguration pattern, which facilitates component composition. A Wire object connects a Producer component and Consumer component service. Data that a source component produces flows through an event chain toward a sink component. A wire also supports advanced features such as filter-based flow control and data type converters.

Figure 6.3 shows how the main concepts of OSGi (bundle, service and wire) support two different Smart Home scenarios. OSGi bundles embed the services and devices of the Smart Home. These bundles registers OSGi services which provides the main system functionality. Finally, OSGi wires enable the communication between the services and devices of the Smart Homes. These wires manages the object interchange through the Smart Home communication channels.

Furthermore, it is possible to enrich an OSGi framework by means of components available in component discovery networks or by specific extensions for a vertical domain [150, 151]. In particular, Prosys Smart-Home extension¹ provides a set of

¹http://www.prosyst.com/products/osgi_ext_smart.html

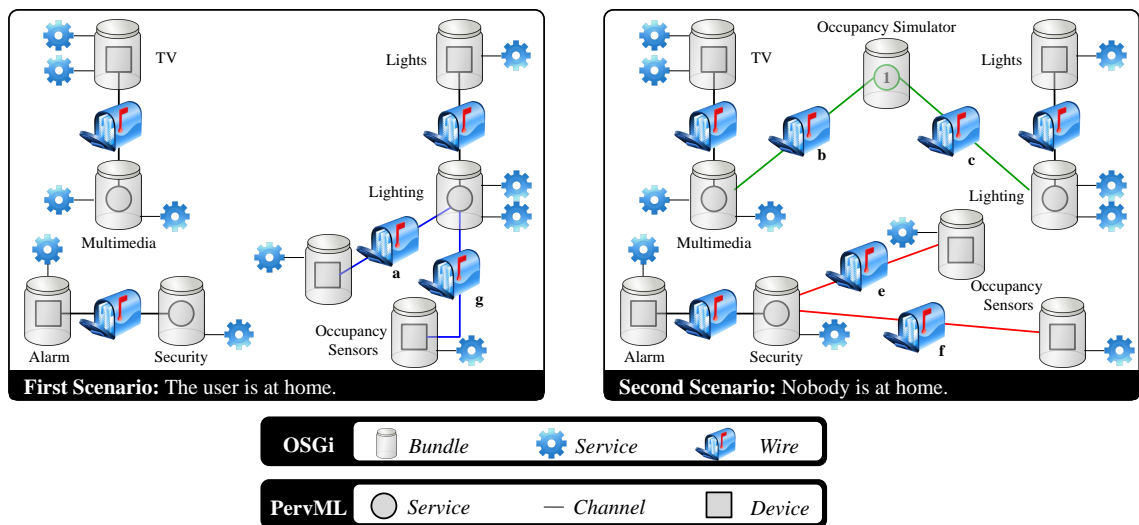


Figure 6.3: The Smart Home from an OSGi perspective.

generic components which enables the development of systems in the smart home domain. This extension covers the following smart home technologies.

- **KNX.** This technology is an standard for applications in home and building control, ranging from lighting and shutter control to various security systems, heating, ventilation, air conditioning, monitoring, alarming, water control, energy management, metering as well as household appliances and audio. This technology can be used in new as well as in existing home and buildings.
- **UPnP.** This technology is a set of networking protocols promulgated by the UPnP Forum. The goals of UPnP are to allow devices to connect seamlessly and to simplify the implementation of networks in the home (data sharing, communications, and entertainment) and in corporate environments for simplified installation of computer components.
- **Bluetooth.** This technology is an open wireless protocol for exchanging data over short distances from fixed and mobile devices, creating personal area networks (PANs). It was originally conceived as a wireless alternative to RS232 data cables. It can connect several devices, overcoming problems of synchronization.

Therefore, OSGi can enable the integration of heterogeneous devices and sen-

sors in pervasive environments such as smart homes. It turns out that the OSGi framework provides not only an infrastructure to implement the *Decentralized Control System Reconfiguration* Pattern but also a portfolio of ready-to-use extensions for pervasive computing applications. Given this reconfigurable architecture, next section shows how to drive architecture reconfigurations by means of Feature models at run-time.

6.3 Reconfiguring the System Architecture through Feature Models

This work suggest that variability models at run-time can assist the system to determine the steps that are necessary to reconfigure its own architecture. In particular, we argue that a system can activate/deactivate its own features dynamically at run-time according to the fulfillment of Context Conditions.

Feature models specify the possible configurations of the system, while a Reconfigurable Architecture can be rapidly retargeted to a specific configuration in response to changes in the context. To achieve this goal, our approach follows the Reconfiguration Process depicted in Figure 6.4.

The first step of our Reconfiguration Process is to feed the Ontology for context modelling with context events. Context conditions check for values in this ontology. For instance, the *EmptyHome* condition is fulfilled when none of the presence detection sensors is perceiving presence. This can be used to trigger the activation of both the *In Home Detection* and the *Presence Simulation* features when all the inhabitants leave home. We can also define another context condition, *Comfort*, to trigger the activation of features related to ease and well-being.

The second step of the Reconfiguration Process is triggered when a context condition is fulfilled. Since a given condition can trigger the activation/deactivation of several features, the Resolution concept (R) represents the set of changes triggered by a condition. A *resolution* is a list of pairs where each pair is

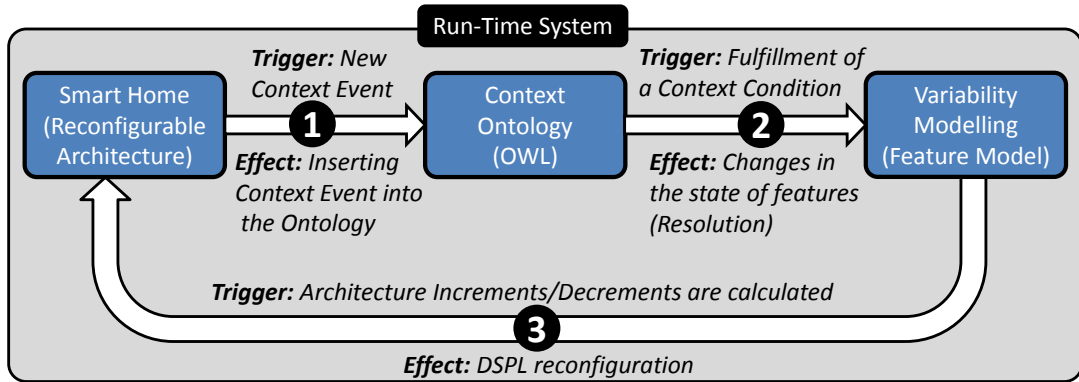


Figure 6.4: Overview of the model-based reconfiguration process.

conformed by a Feature (F) and the state of the feature (S). Each *resolution* is associated to a context condition and represents the change (in terms of feature activation/deactivation) produced in the system when the condition is fulfilled.

For instance, the conditions *EmptyHome* and *Comfort* are associated to the following resolutions:

$$R_{EmptyHome} = \{(OccupancySimulation, Active), (InHomeDetection, Active), (LightingByOccupancy, Inactive)\}$$

$$R_{Comfort} = \{(PipedMusic, Active), (AutomatedIllumination, Active)\}$$

The $R_{EmptyHome}$ resolution means that, when the Smart Home senses that it is empty (condition), it must reconfigure itself to deactivate Lighting by Occupancy and to activate both Occupancy Simulation and In Home Detection.

The third step of the reconfiguration process (see Fig 6.4) addresses the architecture reconfiguration of the Smart Home. In the $R_{EmptyHome}$ example, the Smart Home queries the Feature Model to determine the architecture for that specific context. The architecture increments and decrements are calculated in order to determine the actions to modify the architecture. Specifically, we have defined two operations: **ArchitectureIncrement** ($A\Delta$) and **ArchitectureDecrement** ($A\triangledown$). These operations take a *resolution* as input, and they calculate the modifications to the architecture in terms of Components and Channels.

6.3. Reconfiguring the System Architecture through Feature Models 121

Figure 6.5 shows the Increments and decrements (in terms of both Features and Components) that come from the triggering of a Resolution (R). A Resolution specifies two type of features: (1) features that have to be set to an active state $((F, S) \in R \mid S = \text{Active})$, and (2) features that have to be set to an inactive state $((F, S) \in R \mid S = \text{Inactive})$. The system feature increment is conformed by those inactive features of the current configuration that are indicated as active in the resolution. The system feature decrement is conformed by those active features of the current configuration that are indicated as inactive in the resolution.

By means of the superimposition operation (\odot), it is possible to project a particular feature to the architecture components. If the feature increment is superimposed, some of the resulting components will be in a idle state (quiescent), and the rest of resulting components (labeled as 1 in Figure 6.5) will be in an operational state (active). The architecture increment ($A\Delta$) is conformed by only the resulting components that are in quiescent state.

If the feature decrement is superimposed, all the resulting components will be in an active state. However, it is possible that the intersection between these components and the components of the feature increment is not empty (intersection labeled as 2 in Figure 6.5). That is, some of the components of the feature decrement might be used by the active features of the feature increment. Therefore, the architecture decrement ($A\nabla$) is conformed by only those components of the feature decrement that are not required by the feature increment.

We define $A(\Delta\nabla)$ operations below by means of the *superimposition* (\odot) operator and the *relative complement* (\setminus) operator, (also known as the set-theoretic difference).

$$\begin{aligned} A\Delta &\stackrel{\text{def}}{=} \odot((F, S) \in R \mid S = \text{Active}) \setminus \odot(CC) \\ A\nabla &\stackrel{\text{def}}{=} \odot((F, S) \in R \mid S = \text{Inactive}) \setminus \odot((F, S) \in R \mid S = \text{Active}) \end{aligned}$$

For example, the results of these operations, given $R_{\text{emptyHome}}$ of the First Reconfiguration Scenario (see right side of Figure 6.3), are as follows:

$$\begin{aligned} A\Delta_{\text{emptyHome}} &= \{c, 3, d, e, f\}, \\ A\nabla_{\text{emptyHome}} &= \{a, b\} \end{aligned}$$

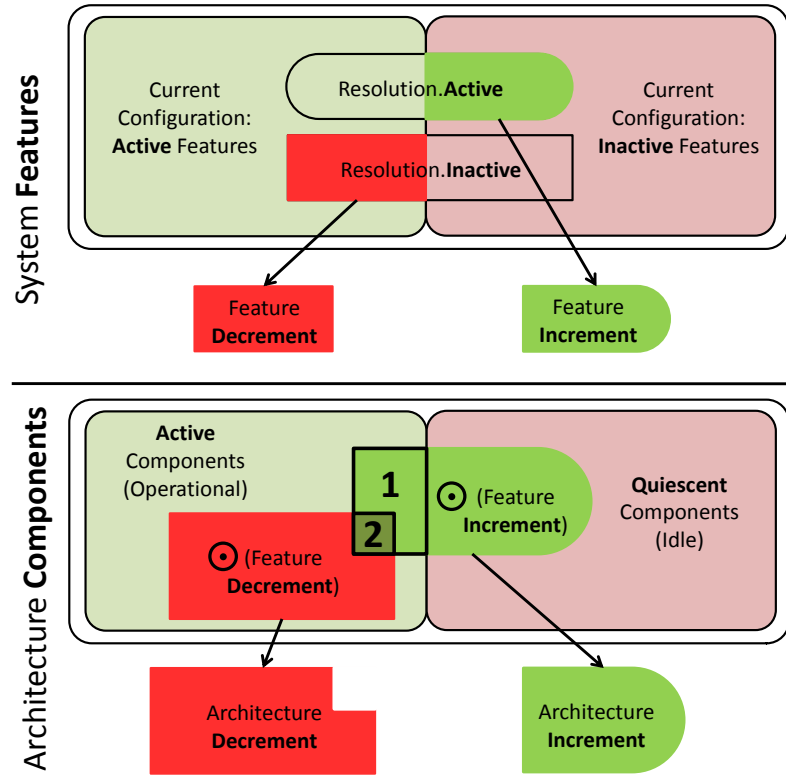


Figure 6.5: Architecture Increment and Decrement given a Resolution.

These $A(\Delta\nabla)$ indicate how system components should be reorganized for the reconfiguration in order to move from one configuration of the system (User at Home, see left side of Figure 6.3) to another configuration (Nobody at Home, see right side of Figure 6.3). As illustrated in Figure 6.3, the occupancy sensors are no longer used for lighting (communication channels a and b are disabled, as indicated in $A\nabla_{EmptyHome}$), and they are used for providing information to the security service instead (communication channels e and f are enabled, as indicated in $A\Delta_{EmptyHome}$). In addition, the occupancy simulator (labelled as 1) is activated, and the communication channels required for this service to communicate with multimedia (channel c) and lighting (channel d) are established as $A\Delta_{EmptyHome}$ indicates.

In this section, we have illustrated how the autonomic reaction of a system can be calculated by taking the variability models as a basis. In the next section, more detail is provided about how the required steps are supported by MoRE, our reconfiguration engine.

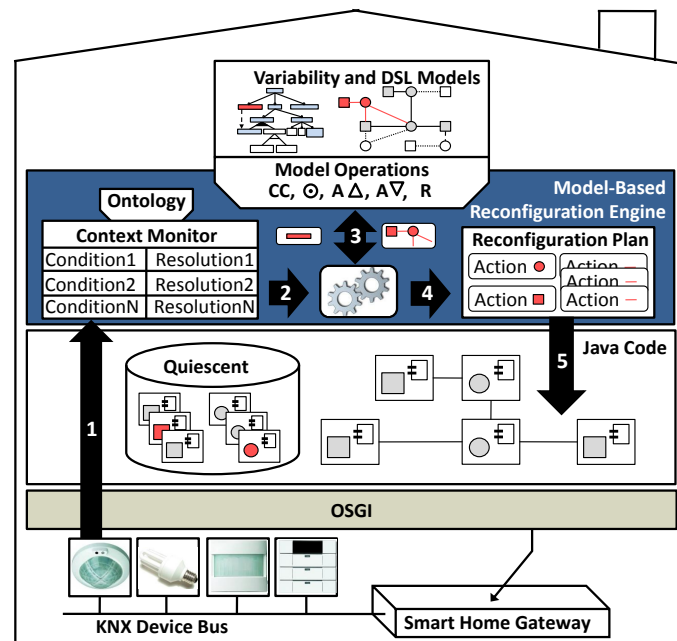


Figure 6.6: The model-based reconfiguration process overview.

6.4 MoRE: Model-based Reconfiguration Engine

To enable autonomic behaviour, the system must evolve from one configuration to another by itself. Since the reconfiguration in our approach is performed in terms of features, a Model-based Reconfiguration Engine (MoRE) is provided to translate context changes into changes in the activation/deactivation of features. Then, these changes are translated into the reconfiguration actions that modify the system components accordingly.

The overall reconfiguration steps are outlined in Figure 6.6. The Context Monitor uses the run-time state as input to check context conditions (step 1). If any of these conditions are fulfilled (e.g., home becomes empty), then MoRE uses the associated resolution and the previous **Model Operations** to query the run-time models about the necessary modifications to the architecture (step 2). The response of the models is used by the engine to elaborate a Reconfiguration Plan (step 3). This plan contains a set of **Reconfiguration Actions**, which modify the system architecture and maintain the consistency between the models and the architecture (step 4). The execution of this plan modifies the architecture in order to activate/deactivate the

features specified in the resolution (step 5). The Quiescent Catalogue contains the inactive components of the system (e.g., drivers of devices that are not in use). Therefore, the devices considered in the variability scope can be incorporated in the system at any time.

6.4.1 MoRE Model Operations

Our proposal makes an intensive use of models. Context events and system variability are represented by models. Context events are represented by means of OWL ontologies, and system variability is captured by means of feature models. For performing the system reconfiguration, information is extracted from these models. Different model query technologies are used at run-time by MoRE depending on the modes involved. MoRE uses SPARQL for OWL manipulation and Eclipse Model Query for Feature Model manipulation. This section introduces the role that both SPARQL and Eclipse Model Query play in MoRE.

Operations for Context Models

Context conditions (e.g., the home being empty) are specified as SPARQL queries to our ontology. SPARQL is the W3C recommendation query language for RDF. This query language is based on graph-matching techniques. Given a data source, a query consists of a pattern which is matched against the data source, and the values obtained from this matching are processed to give the answer. The data source to be queried can be an OWL model as is the one of our ontology for Smart Home context.

A SPARQL query consists of three parts. (1) The pattern matching part, which includes several features of pattern matching of graphs, like optional parts, union of patterns, nesting, filtering (or restricting) values of possible matchings, and the possibility of choosing the data source to be matched by a pattern. (2) The solution modifiers, which once the output of the pattern has been computed (in the form of a table of values of variables), allows to modify these values applying classical operators like projection, distinct, order, limit, and offset. Finally, (3) the output of a SPARQL query can be of different types: yes/no queries (*ASK*), selections of

values of the variables which match the patterns (*SELECT*), creation of new triples (*INSERT*), and descriptions of resources (*DESCRIBE*).

By means of SPARQL queries, we have developed the two operations for manipulating our Context model. On the one hand, the *Inserting Context Event* operation is on behalf of keeping track of the Smart Home context events. On the other hand, the *Context Condition* operation is on behalf of evaluating the values of the ontology.

To implement the *Inserting Context Event* operation, we use the *INSERT* form to insert new triples in the RDF graph of the Ontology. Each new triple is in the form of (subject, predicate, object). The subject and object are the ontology objects or individuals and the predicate is a property relation defined by the ontology. For example, Listing 6.1 shows the query to set that a given user is at home.

```

1 INSERT DATA INTO
2 <http://pros.com/Inhabitant>
3 { <http://ontologies.com
4 /SmartHome.owl#John>
5 pros:name ‘‘John’’ ;
6 pros:isAtHome ‘‘true’’ . }
```

Listing 6.1: Example of Inset Event operation

To implement the *Context Condition* operation, we use the *ASK* form to test whether or not a query pattern has a solution. No information is returned about the possible query solutions, just whether or not a solution exists. That is, *ASK* returns a boolean indicating whether a query pattern matches or not. For example, Listing 6.2 shows the query to evaluate the *Empty Home* condition.

```

1 ASK {
2 ?inhabitant rdf:type
3 pros:Inhabitant .
4 ?inhabitant pros:name ?name;
5 pros:isAtHome ?isAtHome .
6 FILTER (?isAtHome =
```

```
7 | \ "true\"^^xsd:boolean) }
```

Listing 6.2: Example of Context Condition

The combination of the previous operations enables MoRE to gather information about the domain that it shares an interface with (*Inserting Context Event* operation), and to evaluate this information (*Context Condition* operation). Then, MoRE can calculate an appropriate reconfiguration as a response to the current situation.

Operations for Variability Models

The resolution associated to the *EmptyHome* condition (see the definition of $R_{EmptyHome}$ in Section 6.3) specifies which features should be activated or deactivated to manage the change in the Smart Home context. To project this resolution to the system architecture, MoRE queries the feature model in order to calculate the $A(\Delta \nabla)$ operations. We have used the EMF Model Query framework (EMFMQ) to define these model operations.

EMFMQ provides an API to construct and execute query statements in a SQL-like fashion (see Listing 6.3). These query statements can be used for discovering and modifying model elements. Queries are first constructed with their query clauses and then they are ready to be executed.

There are two query statements available: SELECT and UPDATE. The SELECT statement provides querying without modification while the UPDATE statement provides querying with modification. Every query statement requires some query clauses. The SELECT statement requires two clauses, a FROM and a WHERE. The former clause describes the source of model elements where SELECT can iterate in order to derive results. The latter clause describes the criteria for a model element that matches.

```
1 | SELECT
2 | FROM modelElements
3 | WHERE condition
```

Listing 6.3: Template for Model Queries

The FROM clause is set to hierarchical iteration by default, which means that for each element in the modelElements collection, the SELECT statement will traverse its contained elements recursively until it reaches the leaves of the containment subtree to find its matching elements.

The final part of a SELECT statement is the WHERE clause along with its condition. This condition will be evaluated at each model element encountered by the FROM clause to determine whether the element matches the criteria of the query. The condition provided to the WHERE clause falls under a specialized condition called an EObjectCondition that is a condition specially designed to evaluate model elements.

We have implemented the model operations of our approach using the above EMFMQ statements. Next, we show the implementation of both CC and \odot operations. These two operations conform the basics to calculate the $A(\Delta \nabla)$. The purpose for the CC operation is to find Features which state is set to Active. This is implemented straight-ahead using the EObjectAttributeValueCondition which is a condition specially designed to evaluate the value held by a model element (see Listing 6.4).

```

1 SELECT statement = new SELECT(
2   new FROM(resource.getContents()),
3   new WHERE(
4     new EObjectAttributeValueCondition(
5       fm.getFeature_State(),
6       new ObjectInstanceCondition(
7         FeatureConfiguration.ACTIVE))));

```

Listing 6.4: Implementation of the CC Model Operation

The \odot operation returns the set of components related to a feature. This operation is performed to the Model Weaving which main concept is the ElementEqual. An ElementEqual has two members: the Left Element and the Right Element. Left Elements are linked to Features while Right Elements are to Components. We use the EObjectReferenceValueCondition to find those ElementEquals which Left Ele-

ment is related to a given Feature (FeatureIDREF). To navigate through the structure of the ElementEqual, we have compose several Query conditions as follows (see Listing 6.5).

```

1 SELECT statement = new SELECT(
2 new FROM(resource.getContents()),
3 new WHERE(
4     new EObjectReferenceValueCondition(
5     new EObjectTypeRelationCondition(
6     mw.getElementEqual()),
7     mw.getEquivalent_Left()),
8     new EObjectReferenceValueCondition(
9     new EObjectTypeRelationCondition(
10    mw.getLeftElement()),
11    mw.getWLinkEnd_Element()),
12    new EObjectAttributeValueCondition(
13    mw.getWRef_Ref()),
14    new StringValue(FeatureIDREF)))));

```

Listing 6.5: Implementation of the \odot Model Operation

Finally, the references to the model elements returned by the superimpose operation enable the system to construct the Reconfiguration Actions. For instance, the CreateChannel action needs the following information of a channel from the PervML model: producer ID, consumer ID and the type of data that will manage the channel (namely Flavour in OSGi terminology).

Combining both CC and \odot , we have developed the $A(\Delta \nabla)$ operations. These operations define the architecture modifications for moving between different configurations. The core of these operations is the set of active features in a Resolution ($A\Delta$) and the set of inactive features in a Resolution ($A\nabla$). Given a Resolution, Listing 6.6 shows how the to get the active features of the Resolution.

```

1 SELECT statement = new SELECT(
2 new FROM(resolution.getContents()),

```

```

3 new WHERE(
4     new EObjectAttributeValueCondition(
5         fm.getFeature_State(),
6         new ObjectInstanceCondition(
7             FeatureConfiguration.ACTIVE)))));

```

Listing 6.6: Implementation of the Resolution-Active Operation

The *relative complement* (\setminus) operator of the $A(\Delta\nabla)$ operations is also implemented by means of EMF-Model query primitives. First, note the equivalence between this operator and the expression based on NOT and IN operators (see Listing 6.7). That is, the set-theoretic difference between SetA and SetB ($\text{SetA} \setminus \text{SetB}$) is the set of all members of SetA that are not members of SetB.

```

1 SetA \ SetB = IN(SetA) and NOT(IN(SetB))

```

Listing 6.7: Implementation of the \setminus Operator

The \setminus operator is not implemented as is by EMF Model query, but both NOT and IN operators are. The IN operator is an EObjectCondition specialization used to test whether a given model element is present in a collection of model elements. The NOT operator is an EObjectCondition that negates the result of evaluation of another EObjectCondition. Listing 6.8 shows how to combine these operators and the above model operations to make up the $A(\Delta\nabla)$ operations.

```

1 A.Increment = IN(Superimposition(Resolution-ActivatedFeatures))
2             and NOT(IN(Superimposition(CurrentConfiguration)))
3
4 A.Decrement = IN(Superimposition(Resolution-InactivatedFeatures))
5             and NOT(IN(Superimposition(Resolution-ActivatedFeatures)))

```

Listing 6.8: Implementation of the $A(\Delta\nabla)$ Operations

Application of Model Operations

Model operations enable model querying at run-time in order to calculate architecture reconfigurations. Figure 6.7 shows how Context Model operations and Vari-

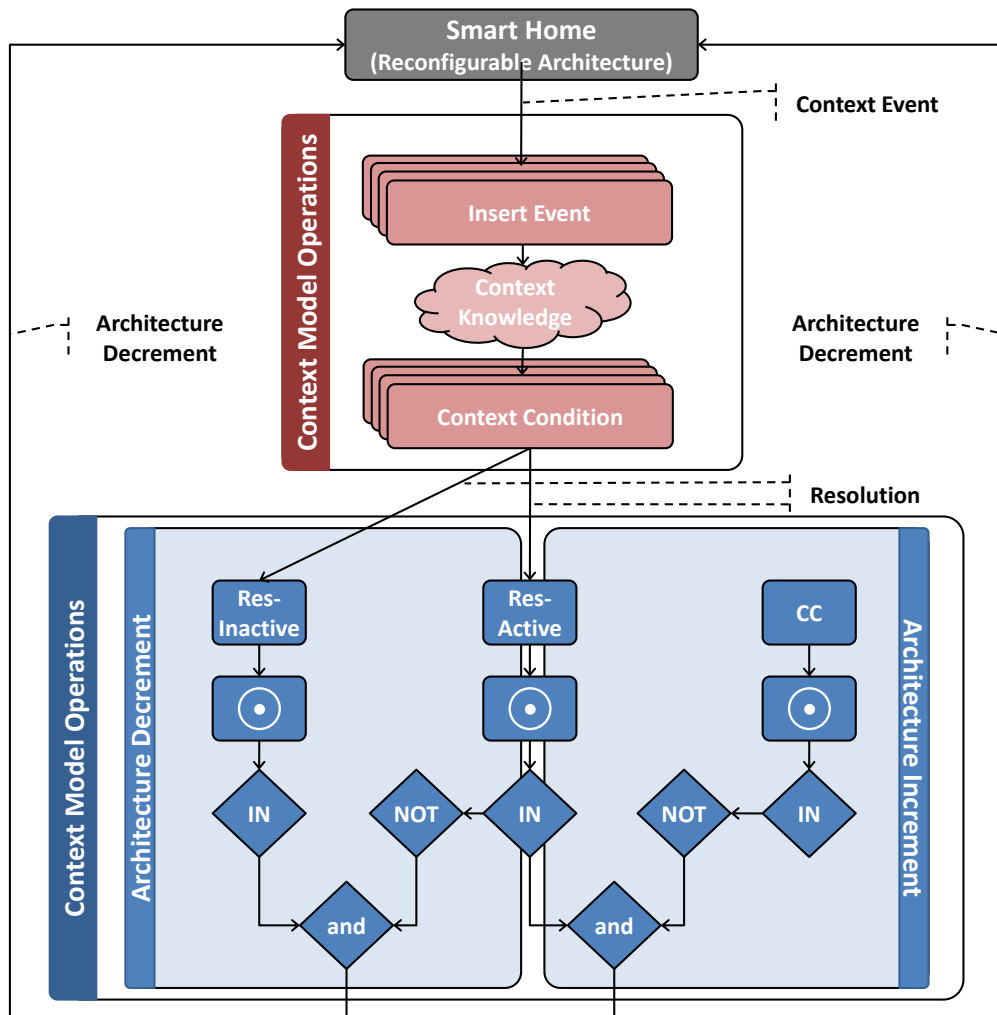


Figure 6.7: Calculating $A(A \Delta \nabla)$ through the Model Operations.

ability Models operations take a context event as input, and they calculate the architecture reconfiguration. Context Model operations manage the occurrence of context events such as *John leaves the Home*. Given this context event, the Insert Event operation sets the new state of the user John in the ontology. Once the ontology is updated, those Context conditions related with the previous event are evaluated. If a context Condition (such as Empty Home) is fulfilled, then the associated resolution ($R_{EmptyHome}$) is triggered.

A Resolution represents the set of changes triggered by a Context Condition in terms of feature activation/deactivation. Variability Model Operations take the Resolution as input (Res-Active and Res-Inactive operations) and they query the Feature Model (CC operation) to determine the architecture for that specific context.

Then, the \odot operation and the *IN* and *NOT* model operators are combined to implement the $A \Delta \nabla$ operations as described in Listing 6.8. The outputs of the operations specified the architecture increments and decrements, which will be used to create the actions to modify the architecture

6.4.2 MoRE Reconfiguration Actions

The reconfiguration of the system is performed by executing reconfiguration actions that deal with the activation/deactivation of components and the creation/destruction of channels among components. Although our general approach is not platform-dependent, we take advantage of the concrete platform to implement the reconfiguration actions. MoRE makes use of the OSGi framework for implementing the reconfiguration actions.

We have developed reconfiguration actions that are classified in three main categories: *ComponentActions*, *ChannelActions* and *ModelActions*. Actions of the two first categories are in charge of reconfigure the Smart Home architecture, while *ModelActions* updates the Feature Model to reflect the new configuration. Reconfiguration actions implement a common Interface (namely *ReconfigurationAction*) which provide an homogeneous way to execute actions with independence of the action category. Specifically, this interface provides the *execute* operation as an entry point to launch the reconfiguration actions. Reconfiguration Action categories are detailed as follows.

Component Actions

This actions enable a component to transit from an active (operational state) to a quiescent (idle) [101] state in order to perform the system adaptation. These quiescent components do not consume processor or memory resources, but they are ready to be started at any time. Therefore, Component Actions keep components that are not relevant for the current configuration in the catalog of quiescent components. Specifically, *StartComponent* actions drive a quiescent component to an active state, and *StopComponent* actions perform the opposite action.


```
1  @Override
2  public void execute() {
3      //Gathering OSGi Services from Component
4      ArrayList OSGiServices = getOSGiServices(componetID);
5      //Sending Quiescent signal
6      Iterator ite = OSGiServices.iterator();
7      while (ite.hasNext()) {
8          OSGiService osgiService = (OSGiService) ite.next();
9          osgiService.sendSignal(Signals.GoToQuiescent);
10     }
11     //Stopping the Component Container
12     Bundle componentBundle = getServiceBundle(componetID);
13     componentBundle.stop();
14 }
```

Listing 6.9: Implementation of the StopComponent Action

Listing 6.9 shows the implementation of the StopComponent action. First, this action gathers those OSGi services related with a particular Component (note that in an OSGi platform component bundles register several OSGi services). For each one of these services, the StopAction sends a *GoToQuiescent* signal. An OSGi service manages this signal according to the reconfiguration pattern, which is presented at the beginning of this chapter. That is, an OSGi service may go through the Waiting For Acknowledgement, Passivating, and Passive state before reaching the quiescent state. This way, impacted components must complete their current computational activity before they can be reconfigured.

For each Service in $A\Delta$, a *StartComponentAction* is created. This action moves a service from the catalogue to the configuration. Services in $A\nabla$ are mapped to *StopComponentActions* which move services from the configuration to the catalogue. As result, only the necessary services for each configuration are running.

Channel Actions

This actions enable a component to establish communication with other components by means of communication channels. Given a new component, Channel Actions register event listeners as services within the OSGi framework. When the new component has an event object to deliver, it calls all event listeners in the service registry. Specifically, CreateChannel actions create a channel between two components, which is implemented by an OSGi wire between a producer and a consumer. Conversely, DestroyChannel actions perform the opposite action.

```
1  @Override
2  public void execute() {
3      //WireAdmin comes with OSGi FW
4      WireAdmin wa = getWireAdmin();
5      Wire[] wires = null;
6      if (wa != null) {
7          //Gathering wires between two given components
8          wires = wa.getWires(producerID, consumerID);
9          if (wires != null) {
10             //Deleting wires
11             for (int pos = 0; pos < wires.length; pos
12                 ++){
13                 wa.deleteWire(wires[pos]);
14             }
15         } else
16             throw new NoWireAdminException();
17     }
```

Listing 6.10: Implementation of the DestroyChannel Action

Listing 6.10 shows the implementation of the Destroy Channel action. First, this action gets the OSGi Wire Admin which manages the interconnecting eventing mechanism of OSGi. Then, those wires between two particular components are gathered in a Wire collection. For each one of these wires, the Wire Admin destroy the event listeners of the channel and keeps the wire buffer alive until it is empty.

Therefore, `DestroyChannelActions` do not interfere with those components that are not impacted by the reconfiguration,

The Channels of $A\Delta$ are created by *CreateChannelActions* which build OSGi Wires between services. While Channels of $A\nabla$ are destroyed by *DestroyChannelActions* which stop the communication between Services, destroying the OSGi Wires.

Model Actions

After the system architecture has been modified, the Feature Model is updated according to the new functionality of the system. This update is performed by means of a partial reflection of the architecture using Model introspection. Model introspection is a powerful feature of existing modelling frameworks like the EMF Model Query. It allows a program to work with any model by querying its structure dynamically at run-time. Model Actions apply this technique to update the Current Configuration of the Feature Model. In particular, the `UpdateFeature` Action sets a particular feature to a given state.

```
1 UPDATE statement =
2     new UPDATE(
3     new FROM(resource.getContents()),
4     new WHERE(new EObjectAttributeValueCondition(
5                 featureModelPackagePackage.eINSTANCE.
6                 getID(),
7                 new StringValue(featureID))),
8     new SET(setFeatureState())
9 );
```

Listing 6.11: Implementation of the Update Features Action

Listing 6.11 shows the implementation of the Update Features action. First, this action is based on the `UPDATE` statement of EMF Model Query. This statement extends the behaviour of the `SELECT` statement to include the `SET` clause that allows some operation to be performed on the result model objects. The result

model element match the *FeatureID* given to the *UpdateFeature*, and its state is set to either active or inactive.

Since the running $A(\Delta \nabla)$ are triggered by a particular Resolution, those features of the resolution must be updated on the Feature model once the architecture reconfiguration is finished. *UpdateFeatureAction* composes the Feature Model with a resolution. The Features of the resolution overwrite the state of the Feature Model and any other feature remains as is in the Feature Model.

Application of Reconfiguration Actions

Reconfiguration Actions provide the basic operations to dynamically change the system architecture. For example, applying the above Reconfiguration Action mappings to the *Architecture*($\Delta \nabla$) of Figure 6.3 will result in the following set of Actions.

$$A_{\nabla \text{ EmptyHome}} = \begin{cases} a \rightarrow \text{DestroyChannelAction}(a) \\ g \rightarrow \text{DestroyChannelAction}(g) \end{cases}$$

$$A_{\Delta \text{ EmptyHome}} = \begin{cases} 1 \rightarrow \text{StartComponentAction}(1) \\ b \rightarrow \text{CreateChannelAction}(b) \\ c \rightarrow \text{CreateChannelAction}(c) \\ e \rightarrow \text{CreateChannelAction}(e) \\ f \rightarrow \text{CreateChannelAction}(f) \end{cases}$$

Furthermore, the Feature Model must also be updated in order to set the feature states of the resolution. The following action updates the Feature Model: *UpdateFeatures*($R_{\text{EmptyHome}}$).

In the example in Figure 6.3, when the users leave home, the system architecture is reorganized to give priority to security. Given the context state as input, MoRE is in charge of composing the suitable actions to perform this change in the architecture. First, MoRE identifies the resolution that is associated to the fulfilment of the *EmptyHome* condition. This resolution (see the definition of $R_{\text{EmptyHome}}$) specifies which features should be activated or deactivated to manage the change in the Smart Home context. In this case, *OccupancySimulation* and *InHomeDetection*

features must be activated whereas the *LightingByOccupancy* feature must be deactivated. This resolution is then projected to the system architecture by applying the superimposition operator. MoRE queries the feature model in order to obtain the $A(\Delta \nabla)$. These $A(\Delta \nabla)$ define the architecture modifications for moving from a *User at Home* configuration to a *Nobody at Home* configuration as illustrated in Figure 6.3. As a result of this reconfiguration, movement sensors in the house are no longer used for the purpose of lighting; they are used for detecting intruders.

MoRE applies different reconfiguration actions to transit from the original configuration to the new one. To achieve the $A\Delta$, a *component action* is applied in order to (1) find the components of the Occupancy simulator service in the catalogue of quiescent components and (2) start these components. Thus, the components are moved from the catalogue to the current configuration of the architecture. The Occupancy Simulator generates inputs for the Multimedia and Lighting services with the aim of deterring thieves by acting as if there were people at home, so *channel actions* are required to connect these services. Additional channel actions are required to connect the movement sensors with the security service. To achieve the $A(\nabla)$, the channels between the Movement sensors and the *Lighting Service* are destroyed in order to deactivate the *LightingByOccupancy* feature.

Once the architecture has been successfully modified, the Feature Model must be updated accordingly. The *LightingByOccupancy* Feature is set to inactive while both *OccupancySimulation* and *In-HomeDetection* are set to active by a *model action* in order to reflect the current state of the system. As a result, both the Feature model and the system architecture are synchronized and support the desired behaviour when nobody is at home.

Aggregating Reconfiguration actions in a Reconfiguration Plan

The consistence between the architecture and the Feature Model is critical, since an unsynchronized model would drive to reconfiguration failures. For instance, if we query the feature model about the $A(\Delta \nabla)$ of deactivating just the *OccupancySimulation* feature (see Figure 6.3), it will reply with one component (labeled as 1) and two channels (labeled as b and c). Once that the component is set to a

quiescent state and the channels are destroyed, the feature *Occupancy Simulation* must be set to inactive in order to reflect the real state of the system. Otherwise, the next time that we query the model, it will reply incorrectly as long as the model is not synchronized with the system.

Activating or deactivating a feature involves performing a set of operations over components and channels, and the entire set must be reconfigured with no exception. Features are defined as atomic units of functionality, so it is not acceptable to activate or deactivate partially a feature.

```
1 public class ReconfigurationPlan {
2     ...
3     public void addReconfigurationAction(ReconfigurationAction
4         action) {
5         ... }
6     public void execute() {
7         int i = 0;
8         try {
9             for (i = 0; i < plan.size(); i++) {
10                ReconfigurationAction
11                    reconfigurationAction = (
12                    ReconfigurationAction) plan.get(i);
13                reconfigurationAction.execute();
14            }
15        } catch (Exception e) {
16            for (int c = 0; c < i; c++) {
17                ReconfigurationAction reconfigurationAction = (
18                    ReconfigurationAction) plan.get(c);
19                reconfigurationAction.rollback(); }
20        }
21    }
22 }
```

Listing 6.12: Implementation of the Reconfiguration Plan

To meet these consistency and atomic requirements, Reconfiguration Actions are aggregated in a Reconfiguration Plan. This plan contains all the reconfiguration actions related to a specific resolution. Therefore, if the execution of a specific action fails, the plan can rollback previous actions within the same resolution (see Listing 6.12).

For the purpose of this work, rollback actions are defined to reverse the system state to the point in which it was before the application of the resolution. That is, for each performed reconfiguration action a complementary action it is performed. For instance, the complementary action of StartComponent is StopComponet and the other way around. Therefore, reconfiguration Plans provide an “all-or-nothing” proposition stating that the set of actions within a resolution must be completed in their entirety or take no effect at all.

However, for some actions with collateral effects (e.g., sending an SMS) compensatory measures are required since they cannot be easily rolled back, but the application of compensatory actions requires further research and falls out of the scope of the present work.

6.5 Scalability Evaluation of Model-management Technologies at Run-time

Since our approach is mainly focused on reuse, we have decided to use the same model representation at run-time that we use at design time. In this way, the need for the definition of technological bridges between design and run-time is avoided. Thus, effort is saved since there is no need to develop these bridges and validate their correctness. Furthermore, this decision has enabled us to reuse the technologies from the Eclipse Modelling Project to implement the model operations in MoRE.

However, model manipulation at run-time (as opposed to design-time) is still subject to the same efficiency requirements as the rest of the system because the execution of the model operations impacts the overall system performance. We are interested in analyzing to what extent system performance could be affected by keeping at run-time the technologies for model manipulation and representation

that are used at design time (XMI in our case).

The execution of these model operations impacts the overall system performance. In particular, the incorporated latency is determined by (a) *the model manipulation Frameworks*, (b) *the model population* and (c) *the metamodel* (which defines the model schema). In this section, we evaluate the performance of manipulating models at run-time using EMF and EMF Model Query. Specifically, we demonstrate the feasibility of using at run-time the models introduce in our approach.

Experimental Setup The target platform used in our experiments is the open source implementation of OSGI Equinox Release 4. To run the instance of Equinox, we used a host with an Intel Core 2 Duo 2.0 GHz processor and 2 GB RAM with Windows Vista SP1 and Java 1.6.0_7 installed. EMF 2.4 and EMF Model Query 1.2 were deployed in Equinox as plug-ins.

When evaluating the running example of the Smart Home, the performance penalization introduced by model processing was not significant. However, in order to validate whether our proposal scales to large systems, we quantified this overhead for large models that were randomly generated.

For evaluation, we used the following randomly generated models: a MOSKitt Feature Model (metamodel version 1.0), AMW (metamodel version 2.0) and PervML (metamodel version 1.0). These models started with one element and they were populated with two hundred new elements each iteration. After the model population, the following model operation were performed: GetFeatureByName, CurrentConfiguration, Superimposing, GetComponentByID and UpdateFeature. The four first operations are the suboperations performed to calculate the $A(\Delta \nabla)$. The UpdateFeature is user by the model actions in order to set a feature state to active or inactive.

Analyzing in detail the experimentation results (see Figure 6.8), we notice that both CurrentConfiguration and GetFeatureByName get a similar time response. These operations are implemented using the EObjectAttributeValueCondition of EMF Model Query. This condition exhaustively navigates the model for elements that fulfill the value condition. In the case of CurrentConfiguration, this is necessary because we are interested in all the features which state is Active. However, we

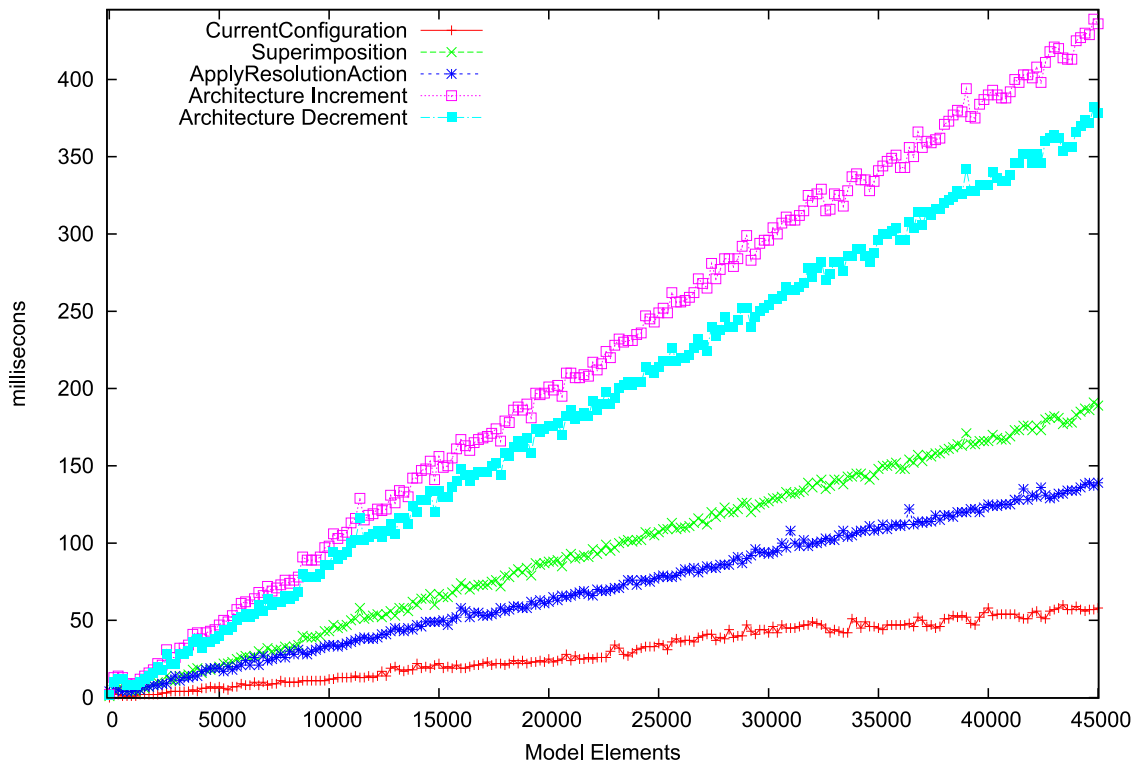


Figure 6.8: Experimental results.

can optimize the `GetFeatureByName` execution. As long as we can guarantee the uniqueness of feature names, the operation just has to search for the first instance with the given name.

Although, the `UpdateFeature` operation also navigates the whole Feature Model to set the state of features, this operation gets significant different results compared to `GetFeatureByName` and `CurrentConfiguration`. This is because `UpdateFeature` has to make persistent the model changes. This operation is implemented by a `UPDATE` statement of the EMF Model Query and a call to the `save resource` of the EMF API.

The `Superimposing` operation gets the worst time response. This is because of the AMW metamodel. This metamodel specifies links between models by means of two indirection levels. To get a component linked to a feature, first the operation has to navigate from a `ElementEqual` metaelement to a `LefElement` metatelement. Second, from this `LefElement` the operation has to navigate to a `ElementRef` metaelement. Furthermore, these two steps have to be performed for each link between a feature

and component. On the other hand, the `GetComponentById` operation gets the best time response. This is because `Superimposition` returns the XMI IDs of the components. In fact, `GetComponentById` is implemented by means of the `XML-Resource Class` of EMF. This class is in charge of serializing and deserializing the models to XML files. Therefore, resolving a component by ID gets best time results.

Overall, even with a model population of 45000 elements in each model, the model operations provide a time response (< 500 milliseconds) that can be considered fast in the Smart Home domain that we are addressing. It turns out that our approach gathers the necessary knowledge from the run-time models without drastically affecting the system response.

The response time offered by the model manipulation operations is acceptable when compared to the performance of the devices and communication networks usually found in the Smart Home domain. Thus, we can conclude that this reuse-based approach can also be applied in other domains with similar temporal constraints.

6.6 Conclusions

In this chapter, we have presented an approach based on Variability Models to guide the choice of system variants in response to context changes in an autonomic manner. In order to support the proposal, a Model-based Reconfiguration Engine (named MoRE) was developed.

MoRE provides model operations that return the necessary $\text{Architecture}(\Delta \nabla)$ in order to move the architecture from one configuration to a new one. The model operations have been implemented in MoRE by means of technologies such as SPARQL and EMFMQ.

MoRE also provides model reconfiguration actions that provide the basic operations to dynamically change the system architecture. The reconfiguration actions have been implemented in MoRE by means of techniques such as the whiteboard and quiescent patterns.

In our experiments, we used an XMI model at run-time in order to determine how to query and update it using the widespread tools of the Eclipse Modelling

Project. Our experimentation shows the feasibility of this approach from the point of view of performance in the Smart Home domain.

We consider that the techniques applied for the Smart Home domain can also be applied to other mass-production environments with similar results. Whether in smart homes, mobile devices or automotive systems, end-users require more and more autonomic functionality.

Next chapter, evaluates a set of alternative strategies for implementing the model operations of MoRE. Results show that the proposed strategies provide the same reconfiguration service with significant differences in quality-of-service.

Chapter 7. STRATEGIES FOR VARIABILITY TRANSFORMATION AT RUN-TIME

“Rien ne se perd, rien ne se crée, tout se transforme.”

– Antoine Lavoisier (1743-1794).

7.1 Overview of the Chapter

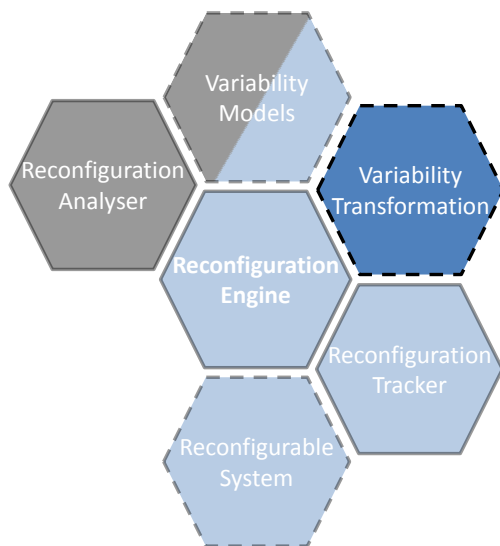


Figure 7.1: Scope of Chapter 7

In this work, variability models are used at run-time to specify the possible configurations of a system, while a Reconfigurable Architecture is rapidly retargeted to a specific configuration. That is, variability models at run-time determine the steps that are necessary for the migration of a software system from one configuration to another. This operation is commonly known in Software Product Line (SPL) community as Variability Transformation, and it is implemented in the core of our Model-Based Reconfiguration Engine (MoRE).

This chapter proposes a set of alternative strategies for implementing the variability transformation. These strategies implement the same reconfiguration functionality but they have different extra-functional properties. For instance, they do not offer the same performances. These strategies enable SPL engineers to set up

MoRE with the most suitable strategy for each concern. For example, MoRE can use a strategy with debugging support as long as the system is under development. When the development is finished and the system is going to be deployed, MoRE can use another strategy with better performance (but without debugging support).

First, this chapter presents the challenges identified in the transition from design variability transformation to run-time variability transformation. For run-time variability, we have applied existing variability modelling approaches based on Feature Models [152] and Fragment Substitutions of the Common Variability Language (CVL) [121]. On the one hand, Feature Modelling provided us with a simple and clear way to visualize variability (which is relevant for design time). On the other hand, Fragment Substitutions enabled us to realize the variability transformation by means of strategies with different extra-functional properties (which is relevant for run-time).

Second, we present the different strategies for run-time variability transformations with significant differences in quality-of-service. Furthermore, we show how these strategies were implemented and validated. Then, we detail the comparison criteria used to evaluate the proposed strategies.

Finally, we present the application of these strategies on the smart-home case study, and we also give recommendations to use the most suitable strategy for different concerns of run-time reconfiguration.

7.2 From Design Variability to run-time Variability: Challenges

A fundamental principle of SPLs is variability management, which involves separating the product line into three parts: common components, components common to some but not all products, and individual products with their own specific requirements. Variability management also involves the managing of the former parts throughout development. In fact, this management has often widespread impact on multiple artifacts in multiple lifecycle stages, making it a predominant engineering challenge in software product line engineering (SPLE).

In traditional SPLE approaches such as Pure::Variants [153], Gears [154, 155] or PLUM [156], variability is mainly managed at design time using configuration and building tools to set compile time variables and select variants of assets. These approaches focus on the development of statically configured systems using core assets with variation points. That is, all variations are instantiated before the system is delivered to customers, and once the decisions are made, they are hard to be altered.

In emerging domains such as self-healing systems [102, 103, 55], context-aware computing [4, 5], and ubiquitous computing [6, 7], software is becoming increasingly complex with extensive variation in both requirements and resource constraints. In addition, modern computing and network environments demand a higher degree of adaptability from their software systems. Computing environments, user requirements, and interface mechanisms between software and hardware devices such as sensors can change dynamically during run-time.

Because it is not feasible to foresee all the functionality or variability that the above systems require, there is a need for Dynamic SPLs [18] (DSPLs) that produce software capable of adapting to fluctuations in user needs and evolving resource constraints. DSPLs bind variation points at run-time, initially when software is launched to adapt to the current environment, as well as during operation to adapt to changes in the environment.

Such DSPLs intensively use variability transformations at run-time in order to adapt their configuration to a changing context and environment. For instance, in the running case study of the smart-home, the system has to be able to accommodate with different context scenarios, such as an empty home or a home with several users within it.

Although dynamic software product lines build on the central ideas of SPLs, there are also differences. For example, the focus on understanding the market and letting the SPL drive variability analysis is less relevant to DSPLs, whose primary goal is to adapt to variations in individual needs and situations rather than market forces.

DSPLs can benefit from modern middleware platforms and programming lan-

guages which provide mechanisms to manage run-time variability. For example, the Java virtual machine allows loading and unloading code dynamically and component platforms allow loading, connecting and disconnecting component instances. However, these mechanisms are platform-specific as they allow for any kind of changes in the running application. Implementing run-time variability at this level consists of writing ad-hoc scripts to program the adaptation policy which remains possible for small applications but becomes tedious and error-prone as the number of configurations for the system grows.

To overcome this problem, we propose the use of variability models at run-time to describe the variants in which a system can evolve. Variability models provide a richer semantic base for run-time decision-making related to system adaptation. The major advantage of this approach is that Feature Model hides much of the complexity in the definition of the adaptation space for an autonomic system as we stated in Chapter 5. However, DSPL still presents other major challenges.

- Modelling the variability. The problem is similar to variability modelling in traditional software product lines except for the variability transformation part which has to be dynamic instead of static. This implies that the variability transformation process has to be fully automated.
- Modelling the run-time adaptation policies, i.e. which configuration should be used and when should the system adapt. Several approaches have been proposed in the literature to express adaptation policies by means of complex AI optimization algorithms [157]. Finding the optimal formalism for modelling adaptation policies remains an open research question but is out of the scope of this work.
- The safe and efficient migration of the system from one configuration to another. At run-time, the variability transformation not only has to produce a plain model of the configuration to run but it has to carry the migration from the currently running configuration to the new one. This is a major difference with typical static variability transformations. The reconfiguration has to be efficient in order to avoid perturbations in the performances of the application

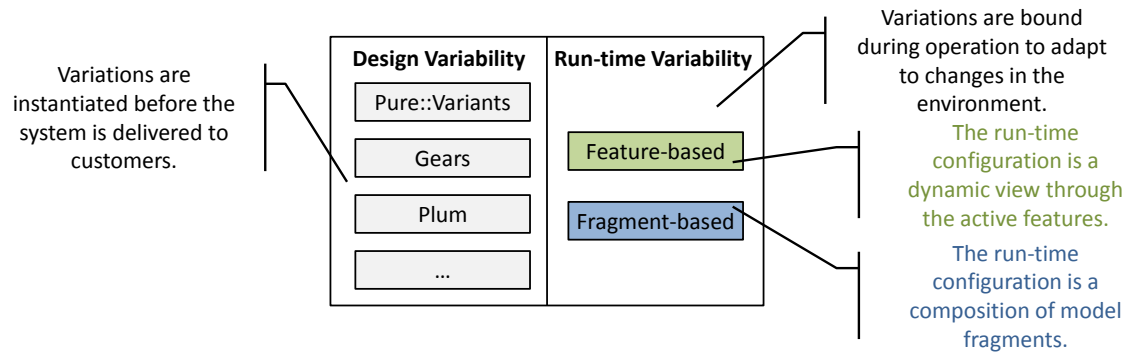


Figure 7.2: Managing Variability.

and it has to be safe, i.e. avoid loss of data and should not disturb the actual services.

Taking these challenges into account, we have evaluated different approaches for managing variability at run-time as next section presents.

7.3 Managing Variability at Run-time

At run-time, existing variability modelling techniques can be applied if they include all necessary links to the domain model in order to automate the variability transformation. As Figure 7.2 shows, these techniques include approaches based on feature-models [152] which relates a domain model to the corresponding features and the Common Variability Language (CVL) [121] which keeps the variability model separated from the base model and expressed the variability by modelling the differences between alternative configurations.

7.3.1 Feature-based Approach

As Chapter 6 shows, we have experimented with the idea of feature models as a dynamic view on a system family model (see Figure 7.3). This view relationship is precisely defined as a mapping characterized by a weaving model. This mechanism can be used for scoping and configuring the system family.

A dynamic feature-based view enables the scoping of a system family model for different context scenarios. The feature model defines a hierarchy of features

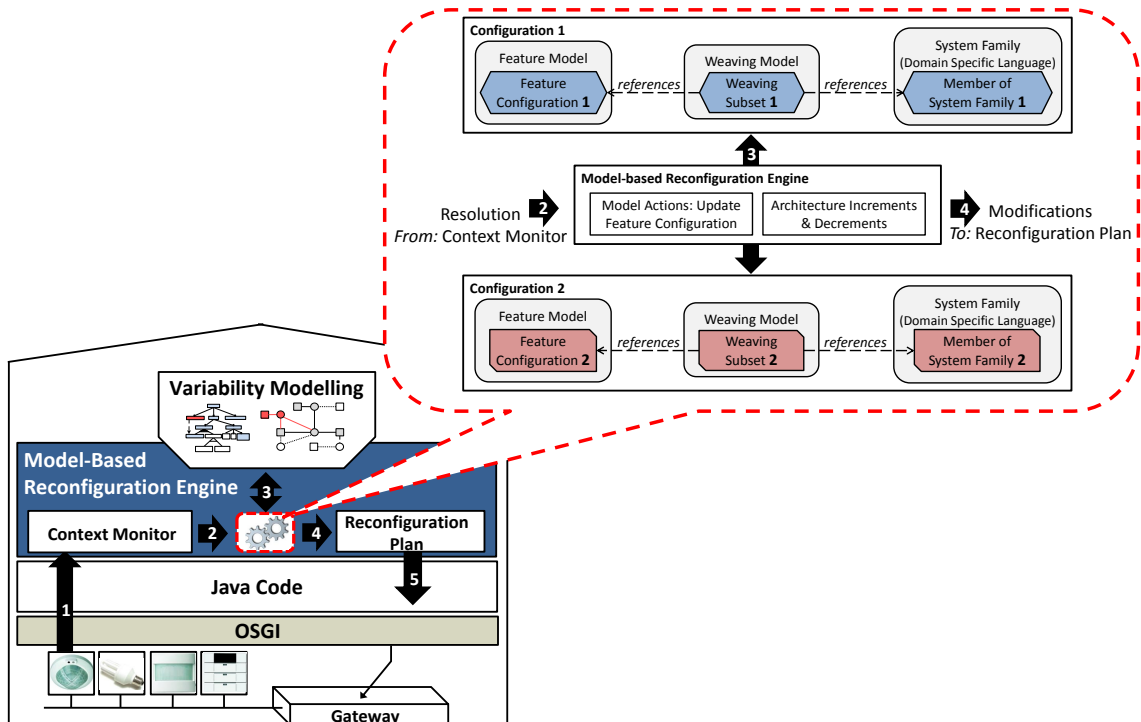


Figure 7.3: Overview of Feature-based superimposition.

together with the constraints on their possible configurations. The system family model contains the union of the model elements that conform the family members. The set of the valid family members (according to the feature model constraints) corresponds to the extent of the system family.

The elements of a particular family member are referenced using a weaving model. The weaving model between a feature model and an system family establishes traceability links between features and family elements. We use this traceability for representing existential dependency constraints. In general, an arbitrary set of feature elements, i.e., features and relationships, are mapped to an arbitrary set of family elements, i.e., services and devices in the smart home example. There is a many-to-many association between feature elements and family elements, but a typical mapping is where an one-to-one mapping is used to express an existential dependency from a feature element to a family element.

This weaving provides traceability between features and their realization in the system family model. In addition, the weaving is evaluated with respect to the current feature configuration, which is determined by the system context. Only

those model elements that are related to an active feature are visible on the dynamic view of the model. That is, the active features through the weaving model indicate whether or not a model element should be present in the current system configuration. Given the duality of the weaving traceability (from features to the system family and the other way around), the weaving can be seen from different perspectives: (1) giving semantics to features in feature models by mapping them to the system family and (2) using feature models to provide a concise representation of variability contained the system family.

7.3.2 Fragment-based Approach

We have also experimented variability transformations based on the model fragment substitutions of CVL. This transformation is domain-independent and therefore it can be applied to any DSL such as the one of the Smart Home (PervML). The Transformation takes a given resolution and then it performs the required fragment substitutions to a base-model in order to synthesises the resulting configuration.

A placement fragment (original) of the base model is the fragment of the model that may be replaced by replacement fragments (alternative). A fragment is defined by a set of boundary elements that give the boundary between the fragment and the rest of the model. When replacing a fragment by another fragment, these boundary elements denote which references between model elements (in terms of meta objects) should be updated in order to have a model according to the metamodel of the base language.

The Fragment Substitution approach expresses the concept of iteration, similar to multiplicity of parts in UML composite structures, and it is even possible to use substitutions to express choice between variants since the substitution may point out more than one model fragment as alternatives and these may be interpreted as separate choices. In the case where a resolution defines more than one chosen alternative for a substitution, this means that a copy of each replacement fragment should be included. The Fragment Substitution concept is also applied to express options. When the Placement has no chosen fragment this means that all the involved objects of the original fragment are simply removed (and the hole closed by

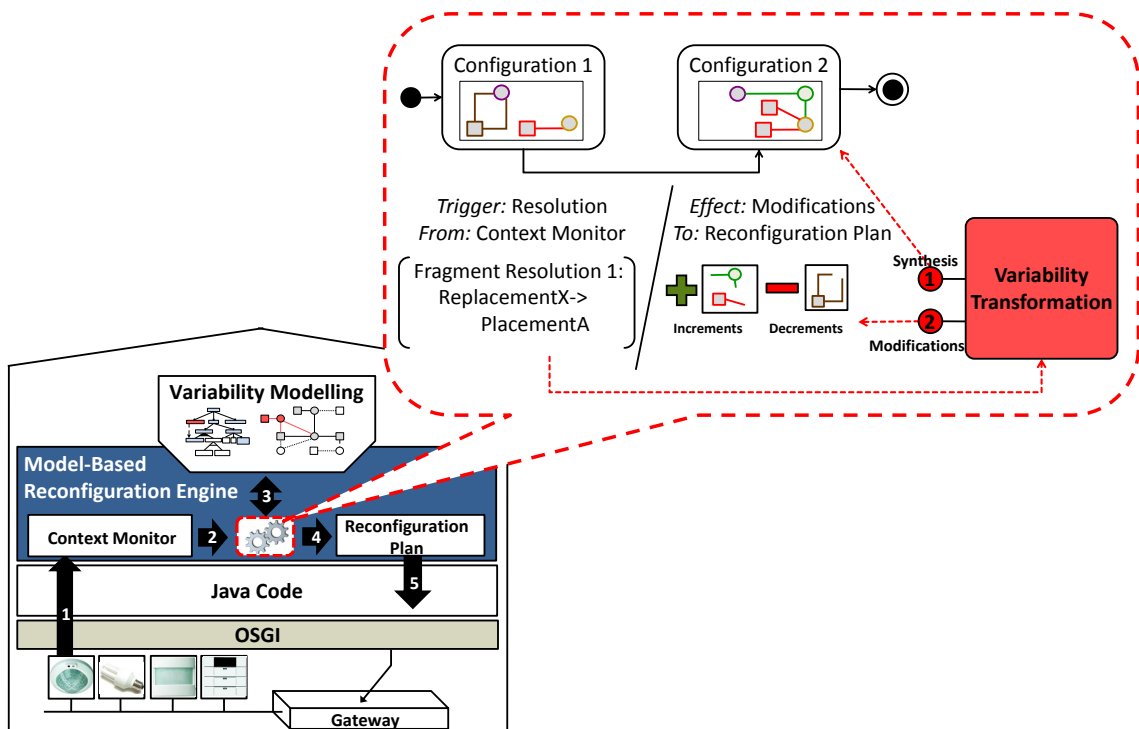


Figure 7.4: Overview of the Variability Transformation.

the replacement boundary elements).

The fragment substitution concept can be seen as a generalization of several of the central feature model concepts and it also incorporates the notion of staged configurations of feature modelling [158].

At run-time, the variability transformation supports the reconfiguration of a base-model from one context scenario to another. Context events are associated to resolutions which drive the fragment substitutions. Specifically, the Variability transformation takes as input a Resolution and then it performs two operations:

1. **Synthesis.** This operation generates a new Base-model configuration that fulfills the given Resolution.
2. **Modifications.** This operation calculates the differences between the previous Base-model configuration and the new Base-model configuration.

Figure 7.4 shows this reconfiguration process using the State machines notation. The states represent different configurations of a Base-model. While the transitions indicate the possibility of Base-model reconfiguration. The reconfiguration of

Figure 7.4 is started by means of a Resolution (trigger of the transition), which specifies the Replacements for each Placement. Given this Resolution as input, the configuration calculates the effects of the fragment substitutions in terms of model increments/decrements (*Modifications* operation). Finally, the transition leads to a state of which Base-model configuration is calculated by the *Synthesis* operation.

The target base-model configuration is calculated by manipulating a copy of the original base model. In this copy of the model, model elements are deleted or populated according to the resolution, resulting in the target configuration. To implement Fragment Substitutions, model element references are modified, and model elements might be moved from one element container to another.

The set of model elements from the base model to be moved are defined by the Replacement Fragment. The transformation processes each boundary and modifies the object structure as follows: for each *toReplacement*, the element referenced by its outside-boundary element is modified to point to the element defined by its binding's inside-boundary element. Correspondingly, for each *fromReplacement*, the element referenced by its inside-boundary-element is modified to point to the element defined by its binding's outside-boundary element.

7.3.3 Assessment between Feature-based and Fragment-based Approaches at run-time

We have successfully applied the above approaches at run-time through the running example of the Smart Home and the Smart Hotel case study (see Appendix A) Based on our experience using these approaches, we provide several thoughts as follows.

Both approaches are general, they work for any model whose metamodel is expressed in the Meta-Object Facility (MOF) [159] or a comparable modelling formalism, and they can be incorporated into existing model editors. From the usability perspective, they are also intuitive. Both features and fragments abstractions successfully enabled us to characterize specific configurations of the systems. However, in both approaches, we required the use of coloring techniques to make easy to see what will be contributed to the DSL model by selecting a given feature or fragment.

Regarding the feature-based approach, a possible concern is that defining the

weaving between features and the system family is not always simple and may require several iterations; however, further tool support can be offered, e.g., for filtering the system family parts relevant to certain features or subset of systems, and automatic verification guaranteeing the well-formedness of all possible family members instances.

In our case, the weaving was explored using a high abstraction-level DSL for smart homes. That is, the feature model and the DSL were at similar levels of abstraction. As a result, despite some complex weavings, most of the mappings were manageable. While the weaving mechanism works for all kinds of mappings, we can imagine, for example, when DSLs are closer to implementation and feature models are closer to requirements, the weaving would become very complex and less manageable.

At run-time, we have successfully applied feature models to scope a system family model for different context scenarios. This enables the system itself to calculate the increments/decrements between different scenarios. However, we mainly recommend feature modelling for the design phase, as it directly shows the impact of selecting a given feature on the resulting model. Feature modelling presents the system designer with a superimposition of all variants whose elements are related to the corresponding features. Therefore, the system designer can activate those features related with a particular context and evaluate the resulting configuration.

Regarding fragment-based approaches, we discovered that there are many relevant variables to perform the fragment substitutions at run-time. For example, we can discard the replaced fragments or keep track of them. In addition, we can work with copy of the fragments or use the same instances always. The combination of the former variables turned out on different strategies to perform the variability transformation at run-time. All these strategies ensure the same reconfiguration service but have different extra-functional properties: for example they do not offer the same performances or they do not offer the same history capabilities.

These strategies enabled us to set up MoRE with the most suitable strategy for each concern, because these strategies cover specific extra-functional requirements such as performance or support to reconfiguration debugging at run-time. Since

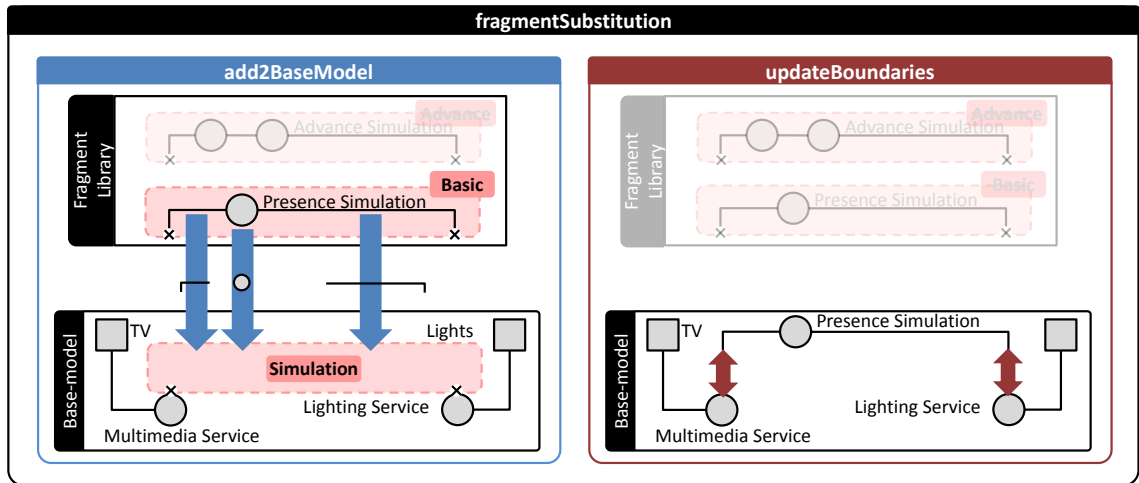


Figure 7.5: Common operations for fragment substitution.

variability transformations are more and more applied to domains which require extra-functional properties, we believe that a catalog of this strategies is also useful for the SPL community that addresses run-time variability.

7.4 Strategies for Variability Transformation

Incrementally, more approaches apply SPLs to build run-time adaptive systems [108, 160, 112]. Although the details are different, these approaches share that they perform the variability transformation intensively at run-time. Furthermore, managing variability at run-time stresses concerns such as performance or reconfiguration debugging. We argue that the variability transformation can be realized by means of different strategies. These strategies implement the same functionality (*synthesis* and *modifications*) but they have different extra-functional properties. For example, they do not offer the same performances. In particular, we have implemented three different strategies as follows.

7.4.1 Common Operations of the Strategies

The strategies for variability Transformation are based on the idea of model fragment substitutions. Given a resolution, the placements of a base-model are dynamically populated with different fragments. This fragment substitution is a common op-

eration that the strategies apply in different manners to implement the variability transformation.

Figure 7.5 shows an overview of the fragment substitution operation. This model operation takes two steps to perform the fragment substitution. First, it gathers the model components that conform the fragment selected by the resolution, and then it adds this components from the fragment to the placement (see left of Figure 7.5). Second, the model operation updated the boundaries of the components added to the placement (see right of Figure 7.5).

Since the strategies for variability transformation are performed at run-time in response to context events, they cannot be manually performed, and must be fully automated. To automate the strategies we have implement them by means of the reflective API of EMF for manipulating Model Elements generically.

Listing 7.1 shows the implementation of the fragment substitution operation. This operation takes as input a replacement fragment, a placement and a boundaries map. The boundaries map specifies the mapping between the fragment and the placement boundaries. In the example of Figure 7.5, boundaries are depicted using an small *x*. For each one of the components that conform a placement, the operation adds this elements to the placement (*add2BaseModel* operation), and updates its boundaries according to the mapping (*updateBoundaries* operation).

```
1  protected void fragmentSubstitution(  
2      ReplacementFragment replacementEObject ,  
3      PlacementFragment placementEObject , Map boundariesMap){  
4      List components = getReplacementComponents(replacementEObject);  
5      Iterator iteComponents = components.iterator();  
6      while (iteComponents.hasNext()){  
7          EObject eObject = (EObject)iteComponents.next();  
8          add2BaseModel(eObject , baseModelResource , );  
9          updateBoundaries(eObject , boundariesMap);}}
```

Listing 7.1: Implementation of the Fragment Substitution

In the context of run-time models manipulated by EMF, a model is defined as a tree structure, as opposed to a directed acyclic graph or just a general graph

with cycles. Except for the model root, every model element is contained by a container element and each contained element knows the element that contains it. The *add2BaseModel* operation takes advantage of the EMF capabilities to query container and contained elements to add model elements from a fragment (which is in a fragment library usually) to a placement (which is in the base-model). This operation queries the base-model about the suitable container for each component of the fragment (see Listing 7.2).

```

1  private void add2BaseModel(EObject eObject, Resource resource){
2      EStructuralFeature containingFeat=eObject.eContainingFeature();
3      EObject container= eObject.eContainer();
4      TreeIterator<EObject> iteResource = resource.getAllContents();
5      boolean found = false;
6      while (!found && iteResource.hasNext()){
7          EObject iteEObject=iteResource.next();
8          if (iteEObject.getClass()==container.getClass()){
9              Object eTargetReference= iteEObject.eGet(containingFeat);
10             if (eTargetReference instanceof EObjectContainmentEList){
11                 EObjectContainmentEList targetList = eTargetReference;
12                 targetList.add(eObject);
13                 found=true;}}}}

```

Listing 7.2: Adding components from a fragment to a placement

Once the fragment components are in the target base-model, the fragment boundaries have to be updated. By means of the reflexion API, the *updateBoundaries* operation (see Listing 7.3) check whether or not a component field is a boundary element. Those boundary elements are updated according to the boundary map.

The above model operations conform the basis of the strategies for variability transformation at run-time. Next, we propose and evaluate three alternative strategies for run-time variability transformations for the migration of an adaptive application from one configuration to another. These strategies have been implemented on top of the fragment-based approach and evaluated on the smart-home case study. Results show that in different situations the proposed variability trans-

formation strategies offer valuable quality-of-service trade-offs. Then, we compared these strategies from the viewpoint of the extrafunctional properties, and we also gave recommendations to use the most suitable strategy for different concerns of run-time reconfiguration.

```

1  private void updateBoundaries(EObject eObject, Map boundaryMap){
2      Field[] fields = eObject.getClass().getDeclaredFields();
3      for (int i=0;i<fields.length;i++){
4          fields[i].setAccessible(true);
5          Object value = fields[i].get(eObject);
6          if (boundaryMap.containsKey(value)){
7              fields[i].set(eObject, boundaryMap.get(value));}}}
```

Listing 7.3: Updating the boundaries of the added components

7.4.2 Regenerative Strategy

Overall, the Regenerative strategy (REG) takes a Resolution as input and it makes a copy of the Base-model which is updated to conform the given Resolution. Figure 7.6 shows the operations of the REG strategy graphically.

In detail, the *synthesis* operation is implemented as follows (see left of Figure 7.6). Given a Resolution (in terms of CVL Fragment Resolutions), first the REG strategy creates a copy of the Base-model (dashed line labeled as *Copy*). Then the strategy iterates all the Fragment Resolutions. Each Fragment Resolution indicates the Replacement Fragment of a Fragment Substitution in the Var-model. For each Fragment Substitution the strategy updates the copy of the Base-model (dashed line labeled as *Update*). In the update of the Base-model copy, those elements referenced by a Placement of each Resolution are deleted, and those elements referenced by a replacement of each Resolution are copied from the Library to the copy of the Base-model. Once all the Fragment Substitutions have been processed, the updated copy of the Base-model is conforming to the given Resolution.

To implement the *modifications* operation (see right of Figure 7.6), the REG strategy calculates the model difference between the new Base-model and the previ-

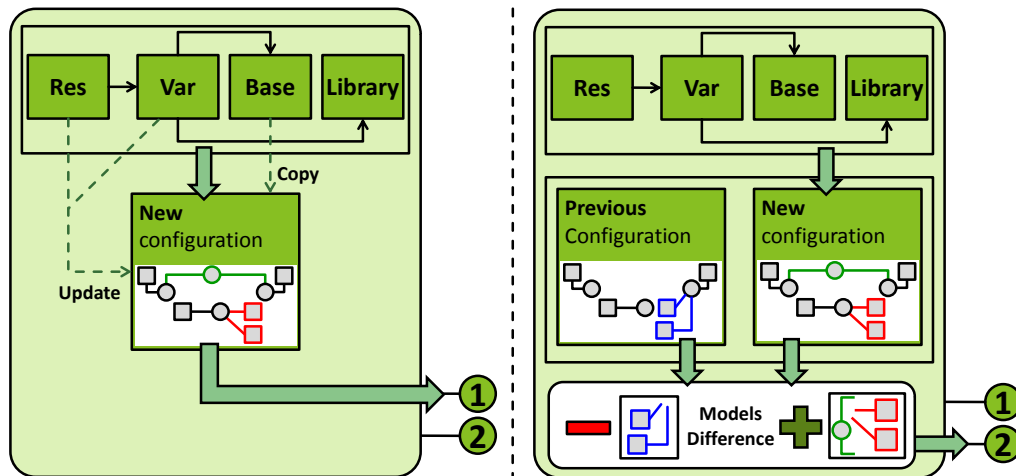


Figure 7.6: Regenerative Strategy.

ous Base-model. The model differences are calculated by means of the EMF Model Compare framework. EMF Compare brings model comparison to the EMF framework, this tool provides generic support for any kind of metamodel in order to compare models.

```

1 public void modelComparison(EObject before, EObject after){
2   Map options = setComparisonOptions();
3   MatchModel match = MatchService.doMatch(before, after, options);
4   DiffModel diff = DiffService.doDiff(match, false);
5   List differences = new ArrayList(diff.getOwnedElements());
6   Iterator ite = differences.iterator();
7   while (ite.hasNext()){
8     DiffElement diffElement = (DiffElement) ite.next();
9     if (diffElement instanceof DiffGroup){
10      DiffGroup diffGroup= (DiffGroup) diffElement;
11      classifyDiff(diffGroup, removedElements, addedElements);}}}
```

Listing 7.4: Implementation of the Model Comparison

The comparison process is divided in two phases: matching and differencing (see Listing 7.4). The matching phase browses the model version figuring out which element comes from which other one, then the differencing process browses the matching result and create the corresponding delta. This delta is a set of *DiffGroups*. Each

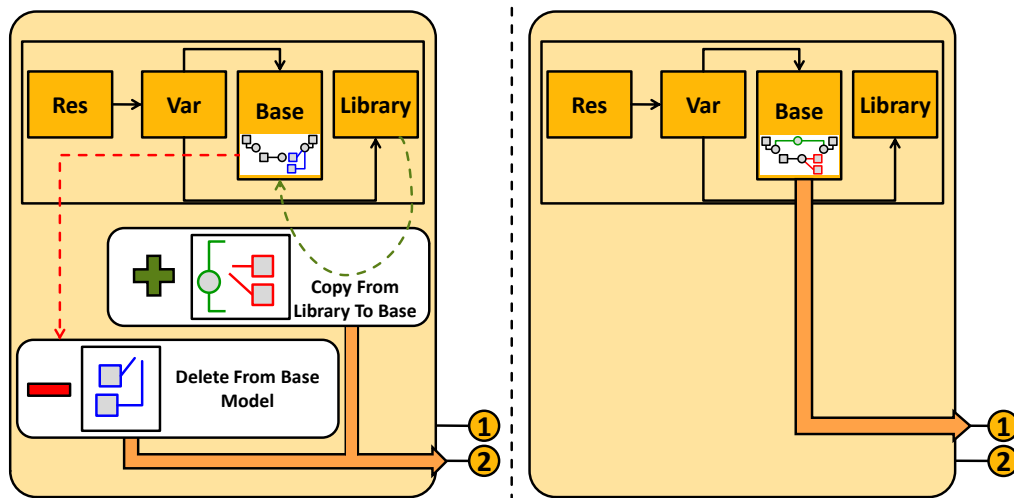


Figure 7.7: Incremental-Copy Strategy.

DiffGroup is used as container for differences which are classified into *removedElements* and *addedElements* between the source and the target configurations.

7.4.3 Incremental - Copy Strategy

Overall, the Incremental - Copy strategy (INC-C) modifies the Base-model of the CVL specification to implement the *synthesize* and *modifications* operations. That is, the strategy does not make a copy of the Base-model. All the required modifications are directly applied to the Base-model of the CVL specification. Figure 7.7 shows the INC-C strategy graphically.

The *synthesis* operation is implemented as follows (see left of Figure 7.7). Given a Resolution, the INC-C strategy iterates all the Fragment Resolutions. For each Resolution the strategy updates the Base-model. Those elements referenced by a Placement are deleted (red dashed line), and those elements referenced by a Replacement Fragment are copied from the Library to the Base-model (green dashed line). Finally the updated Base-model is according to the given Resolution.

To implement the *modifications* operation (see right of Figure 7.7), the strategy iterates all the Fragments of the Resolution. Those elements referenced by a Placement (which should be deleted from the Base-model) are copied to a list of decrements, and those elements referenced by a Replacement (which should be copied from the Library to the Base-model) are copied to a list of increments.

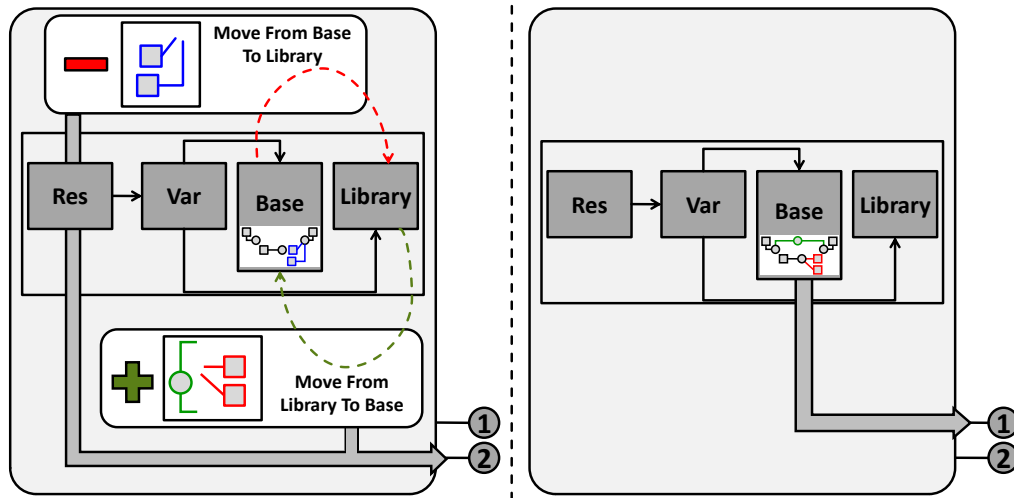


Figure 7.8: Incremental-Move Strategy.

7.4.4 Incremental - Move Strategy

Overall, the Incremental - Move strategy (INC-M) modifies both the Base-model and the Library of the CVL specification. The Library is updated because the model fragments are not removed from the CVL specification. Instead, they are moved from the Base-model to the Library. Figure 7.8 shows the INC-M strategy graphically.

The *synthesis* operation is implemented as follows (see left of Figure 7.8). Those elements referenced by a Replacement Fragment are moved from the Library to the Base-model (green dashed line) and those elements referenced by a Placement are moved from the Base-model to the Library (red dashed line). Therefore, changes performed to the elements of the Base-model are not discarded by reconfigurations, because model changes are saved in the Library.

To implement the *modifications* operation (see right of Figure 7.8), the strategy iterates all the Fragments of the Resolution. Those elements referenced by a Placement Fragment (which should be moved from the Base-model to the library) are copied to a list of decrements, and those elements referenced by a Replacement Fragment (which should be moved from the Library to the Base-model) are copied to a list of increments.

7.4.5 Implementation of the Strategies

We have implemented these strategies by means of the run-time capabilities of the Eclipse Modelling Framework [161]. Specifically, we take advantage of the model manipulation, model compare and model query capabilities of Eclipse Modelling Framework, Eclipse Model Compare and Eclipse Model Query respectively.

For instance, Listing 7.5 shows the implementation of the REG strategy. First, this strategy combines the resolution triggered by a context event with the current resolutions of the system. Then the updated resolutions are processed to elaborate a resolution map. This map specifies the mapping between placements and replacements.

Once the resolution map is calculated, the REG strategy performs a copy of the original Base-model. This copy of the Base-model describes the configuration of the system where no fragment substitution has been performed. On this copy of the Base-model, the strategy runs the fragment substitution operation which is a common operation to all the presented strategies.

Finally, the copy of the base model is modified in such a way that fulfils all the required resolution which include the last resolution triggered by a context event.

```
1 private void executeTransformation() {
2     List currentResolutions = getFragmentResolution();
3     combineResolutions(currentResolutions, contextResolution);
4     Iterator ite = currentResolutions.iterator();
5     Map fragmentSubstitutions = new HashMap();
6     while (ite.hasNext()) {
7         FragmentResolution fragmentResolution = ite.next();
8         Replacement replacement = fragmentResolution.getReplacement();
9         Placement placement = fragmentResolution.getPlacement();
10        fragmentSubstitutions.put(placement, replacement);
11    }
12    Resource originalBaseModel = getBaseModel();
13    Resource copyOfBaseModel = copyModel(originalBaseModel);
14    runFragmentSubstitution(copyOfBaseModel, fragmentSubstitutions);
```

```
14 | saveBaseModelResource ();}
```

Listing 7.5: Implementation of the REG Strategy

Although we have implemented the strategies using the Eclipse Modelling Framework at run-time, the strategies as such can be implemented with other technologies such as ATLAS Transformation Language [162] or MOFScript [163]. For instance, in a previous work [121], the fragment substitution operation is implemented by means of MOFScript.

Next section shows how we have validated the implementation of the above strategies. Then, we describe the extra-functional properties of each strategy and we also give recommendations to use the most suitable strategy for different concerns of run-time reconfiguration.

7.5 Validating the Strategies Implementation

Given a Resolution, we have three different strategies to calculate the same operations (*synthesis* and *modifications*). We argue that simultaneously comparing the outputs of the strategies enables the validation of the strategy implementations.

Our approach tests for equality the operation results of the strategies. In our case, equality means: (1) all the strategies got the same model modifications for each reconfiguration and (2) there are no differences between the resulting base models.

To systematize the process, we perform the testing throughout the Possibility Space of a CVL specification. This Possibility Space is the representation of all the feasible configurations according to the CVL Specification. Top of Figure 7.9 shows a simple CVL specification and bottom of Figure 7.9 shows the Possibility Space using the State Machines Notation. States represent configurations and transitions represent reconfigurations as introduced in the previous section.

Our approach for validating the strategies implementation is a three steps process. First, from a CVL specification we calculate the skeleton of the Possibility Space. This skeleton is conformed by the empty states of the state machine and the transitions with their triggers (Resolutions). An empty state means that the Base-model associated to this state is not calculated yet.

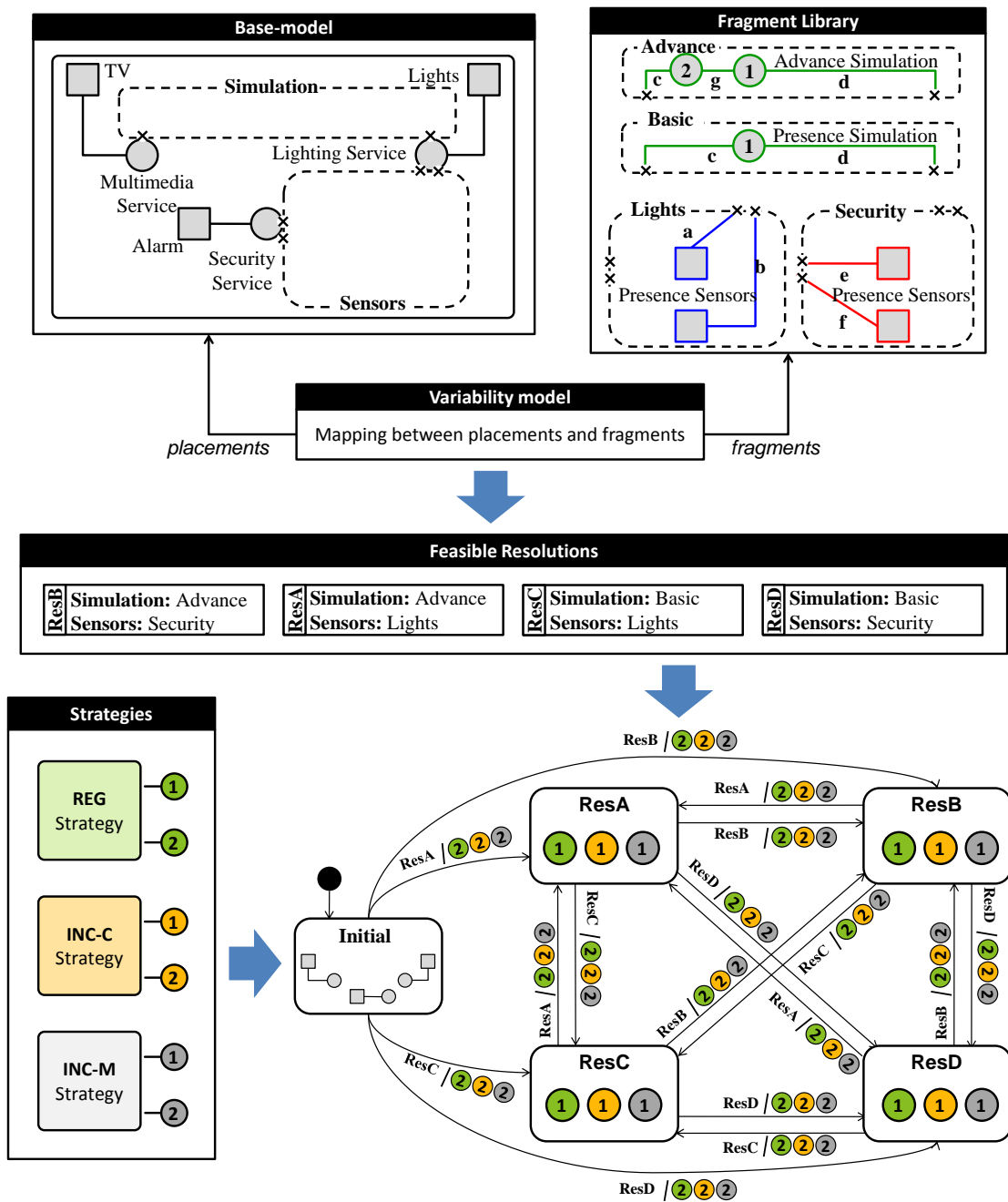


Figure 7.9: Possibility Space.

Second, the Base-model associated to each state and the effects of the transitions are calculated by means of the strategy operations (*synthesis* and *modifications*). For each reconfiguration (transition), the strategies take the same Resolution as input (transition trigger) and they calculate the model modifications (transition effect) and the Base-model associated to the target state.

Finally, our approach compares the model modifications and the Base-models **among strategies** in order to check their equality. We recommend this comparison among strategies when we have at least one reliable strategy and we are implementing new strategies. The Base-model comparison can detect differences between new implementations and a reliable implementation.

Furthermore, some states of the Possibility Space can be reached through different paths (see bottom of Figure 7.9). Independently of the followed path, all the strategies must generate the same Base-model. Our approach also compares the Base-models **among paths** in order to check their equality. In fact, this last comparison can be performed by means of only one strategy.

We recommend this comparison among paths when we do not have a reliable strategy yet. This comparison helps to refine the implementation of a strategy until the inconsistencies among paths have been eliminated.

The combination of these comparisons (among strategies and among paths) throughout a Possibility Space turns out to be a powerful tool to verify the implementation of strategies. We have applied this approach to verify the three strategies presented in the previous section. Furthermore, the approach enables us to validate the implementation of new strategies.

7.5.1 Tool Support for Testing Strategies

Calculating the skeleton of the Possibility Space and then executing and comparing the different strategies are tedious tasks. We have developed a tool to automate this process. Figure 7.10 shows the Testing tool for CVL Strategies, which is integrated with the CVL editor. This tool is structured in three tabs: *Possibility Space*, *Strategies Management* and *Strategies Comparison*.

In the *Possibility Space* tab (see left of Figure 7.10), the testing tool calculates the skeleton of the Possibility Space. To calculate this skeleton, the tool takes as input a CVL specification (see top of Figure 7.10). Then, the tool calculates all feasible Resolutions according to the VAR-model. These Resolutions are valid assignments of Replacement Fragments to Placements. For each Resolution the tool creates an empty state in the Possibility Space. Empty state means that the associated

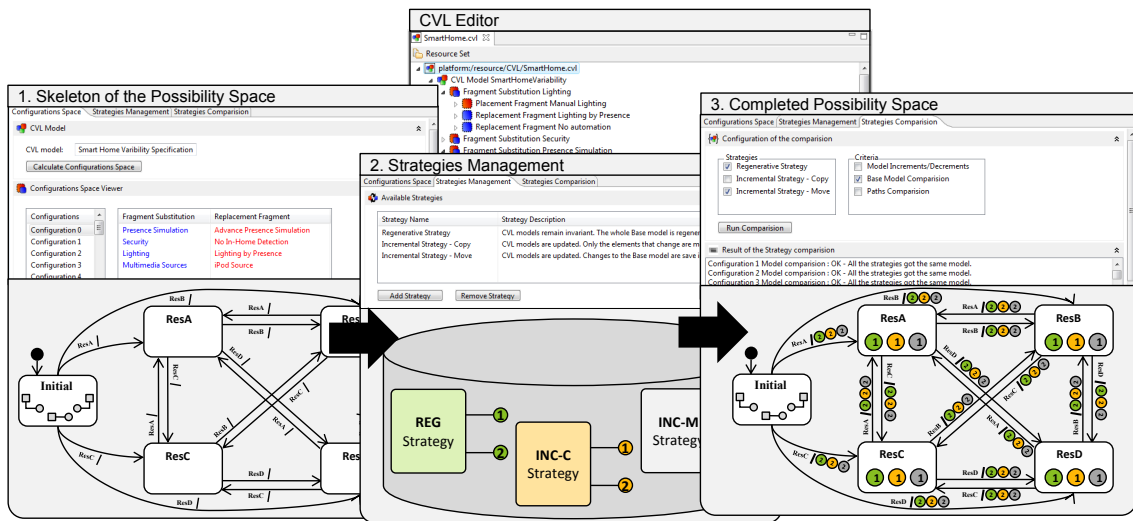


Figure 7.10: Testing Tool for Strategies.

Base-model Configuration will be calculated later by the strategies.

Finally, the transitions are set among states. For each possible pair of states such as *ResA* and *ResB*, the tool creates two transitions: one transition from A to B that is triggered by Resolution A and other transition from B to A that is triggered by Resolution B. Once all the transitions have been set, the skeleton of the Possibility Space is ready.

The *Strategies Management* tab (see center of Figure 7.10) shows all the strategies that are available in the testing tool. The strategies introduced above (REG, INC-C and INC-M) are preloaded in the tool. Furthermore, the tool also provides functionality to load new strategies as Eclipse Plug-ins. These strategies are in charge of completing the skeleton of the Possibility Space.

The *Strategies Comparison* tab (see right of Figure 7.10) runs the testing process. First, the skeleton of the Possibility Space is completed by means of the selected strategies. The *Synthesis* and *Modifications* operations calculate the Base-model for each state and the Effect for each transition. Once the Possibility Space is completed, the tool runs the test for equality.

The criteria for the equality test can be selected by the user among the following options: Model Increments/Decrements, Base-model Comparison and Paths Comparison. The first two options implements the comparison *among strategies* while the last option implements the comparison *among paths*. The tool applies these

criteria thought the Possibility Space to perform the equality test. Finally, the tool generates a report that summarizes the results of the test for equality.

7.6 Extra-Functional Properties of Strategies

Although all these strategies implement the same operations introduced at the beginning of this document, there are differences among them from the viewpoint of the extra-functional properties. Table 7.1 summarizes these differences by means of the following criteria: History, Performance and Persistency of the Base-model changes.

Strategy	History Support	Performance	Persistency of changes
REG	Yes	All Variation Points + Base Model Differences	No
INC-C	No	Only Modified Variation Points	No
INC-M	No	Only Modified Variation Points + Library Update	Yes

Table 7.1: Extra-Functional Properties of the Strategies

The **History Support** criterion evaluates if the strategies keep information about the Base-models that have been previously synthesized. This information is useful for techniques that debug invalid configurations and derive the minimal set of changes to fix flawed configurations [164]. Specially, when these techniques deals with SPLs which use *staged configuration* [165]. Staged configuration means that variability decisions are taken in multiple stages to form a complete configuration iteratively. For instance, the Configuration Understanding and REmedy (CURE) tool implements some of these techniques to debug invalid configurations [166].

By analyzing the introduced strategies from the viewpoint of the History criterion, we found the following results. On the one hand, the REG strategy stores previous configurations of the system, because this strategy works with copies of the

Base-model. On the other hand, INC-C and INC-M just store the current state of the system, since these strategies apply all the modifications to the same Base-model.

Therefore, we recommend the use of the REG strategy for configuration debugging, since the REG strategy will provide more information for the analysis.

The **Performance** criterion evaluates how many elements are processed to synthesize a Base-model configuration. The performance of the synthesize operations is specially important for DSPLs. DSPLs products are adaptive systems, i.e. a product might pro actively adapt itself when changes are performed in its environment. For instance, [109] uses a DSPL to synthesize new system variants for mobile devices according to changes in the context.

From the performance viewpoint, executing the INC-C strategy only involves those elements that change from the previous configuration to the new one. The INC-M strategy additionally updates the Library in order to save Base-model changes. Finally, the REG strategy involves all the variations points, since this strategy regenerates the whole Base-model always.

Therefore, we recommend the use of the INC-C strategy for DSPLs, since in DSPL the performance of the synthesize operation impacts on the overall performance of the synthesized product.

The **Persistency of the base Model Changes** criterion evaluates the capability of strategies to save changes of the Base-model by run-time systems. Increasingly, some approaches are leveraging variability models at run-time [146]. A key benefit of using models at run-time is that models can provide a richer semantic base for run-time decision-making related to system adaptation and other run-time concerns. For instance, [112] leverages variability models at run-time to achieve dynamically adaptive systems.

Analyzing the introduced strategies from the viewpoint of the Persistency criterion, we found that only the INC-M strategy saves changes to the Base-model. In a reconfiguration, this strategy moves the decrements of the Base-model to the Library instead of just deleting these model elements. Eventually, these elements will be back from the Library to the Base-model. The REG and INC-C strategies discard the base model changes, since they just delete the decrements of the Base-model.

Therefore, we recommend the use of the INC-M strategy for approaches that leverage models at run-time. These models can be modified by the run-time system without losing the modifications in the next reconfiguration.

In this section, we have shown three strategies which support different extra-functional properties. Our intent is to develop new strategies that mix the above properties. For instance, INC-C with History support or REG with persistency of base model changes. Furthermore, we also plan to develop more strategies that support new extra-functional properties.

7.7 Applying the Strategies to Smart Homes

We have applied the previous strategies to the running Smart Home case study. This case study allows Smart Homes to use the variability modelling from the SPL design at run-time in order to determine the steps that are necessary to reconfigure the Smart Home. For instance, the Smart Home reconfigurable architecture is retargeted to a *Nobody at Home* configuration when the users leave the home.

This DSPL for Smart Homes use staged configuration since fragments are selected in multiple stages to form a complete configuration iteratively. At a late stage in the configuration process, developers may realize that a specific context condition cannot select some fragments due to reconfigurations in some previous stages. It is hard to debug the last configuration to figure out how to change reconfigurations in previous stages to make these fragments selectable [164]. To debug these *staged configuration errors*, we apply the REG strategy to synthesize the Smart Home configurations. The REG strategy synthesizes configurations from an invariant CVL specification, keeping the history of configurations. Therefore, we are able to conduct a thorough analysis of the previous configurations for the purpose of debugging.

A fundamental problem in SPL engineering is that a real product line can easily incorporate several thousands of variation points [167]. The use of variability models to assist the system adaptation (as our DSPL for Smart Homes does) impacts the product performance. The incorporated latency comes from the synthesizing operation that is performed at run-time when a context condition is fulfilled. For this reason,

we use the INC-C strategy for deployment and we keep the REG strategy just for debugging. The INC-C strategy achieves better performance results since it only involves the subset of variation points affected by a context condition. On the other hand, the REG strategy always involves all the variation points independently of the context condition.

In an ongoing work [168], we also use this DSPL for service reconfiguration in the Ambient Assisted Living domain. In this domain we update the Base-model at run-time in order to save user preferences. The INC-M strategy is suitable for this purpose, since it enables the run-time system to modify the Base-model without losing the modifications in the next reconfiguration.

The realization of the variability transformation by means of interchangeable strategies enables SPL engineers to use the most suitable strategy for each concern. In this section, we have illustrated how to take advantage of different strategies in a DSPL for adaptive Smart Homes. However, we can develop new strategies that provide other extra-functional properties in order to support more SPLs. Furthermore, the tool presented in this chapter will help us to validate the implementation of these new strategies.

7.8 Conclusions

Increasingly, more approaches apply the variability transformation of SPLs to build run-time adaptive systems [160, 112]. We argue that the variability transformation can be realized by means of interchangeable strategies that have different extra-functional properties. These strategies enable SPL engineers to use the most suitable strategy for each concern, because these strategies cover specific extra-functional requirements such as performance or support to reconfiguration debugging at run-time.

In this chapter, we introduced three different strategies (REG, INC-C and INC-M) for realizing the variability transformation. We implemented these strategies by means of the Model Query project of the Eclipse Modelling Framework, and we validated these implementations using a testing approach which provides tool

support. Then, we compared these strategies from the viewpoint of the extra-functional properties, and we also gave recommendations to use the most suitable strategy for different concerns of run-time reconfiguration.

Finally, we have evaluated the above strategies in a SPL for run-time adaptive Smart Homes. In this SPL we illustrated how we have take advantage of the different strategies in practice.

Our intend is to build a catalog of new strategies that cover extra-functional requirements. We believe that this catalog is useful for the SPL community since variability transformations are more and more applied to domains which require extra-functional properties. Furthermore, although the strategies presented in this work are based on CVL, the approach as such, can be applied by means of other languages for variability specification.

Chapter 8. EVALUATION OF THE PROPOSAL

“One of the great mistakes is to judge policies and programs by their intentions rather than their results.”

– Milton Friedman (1912-2006).

8.1 Overview of the Chapter

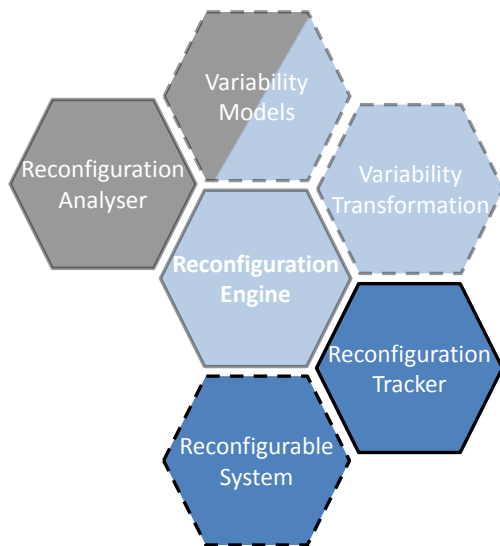


Figure 8.1: Scope of Chapter 8

potential run-time failures can be anticipated during system design [123].

In this chapter, we are concerned with reliability-based risk of run-time reconfigurations, which depends on both the probability that the software product will fail in the operational environment and the adversity of that failure. We have consid-

The software systems achieved in this work are capable of modifying themselves with respect to changes in their operating environment by using run-time reconfigurations. Variability models specify the possible system configurations, while a reconfigurable architecture can be rapidly retargeted to a specific configuration. Since the models that form the basis for run-time reconfiguration are available at design time, it is possible to validate reconfigurations at an early stage of the development process without first implementing them as Chapter 5 shows. However, not all

ered this aspect since it is especially relevant when dealing with Dynamic Software Product Lines (DSPL). For traditional Software Product Lines, once a product is obtained for a given configuration, it can be tested intensively before it reaches the end-users. However, the case of DSPLs is different since different configurations are obtained at run-time. A failure in DSPL reconfigurations directly impacts the user experience since the reconfiguration is performed when the system is already under the user control. Thus, we consider that the risk of run-time reconfigurations must be controlled for reconfigurations that are produced in the operational environment (as is the case of DSPLs). To this end, we have adopted the definition in [169], which defines reliability risk as a combination of two factors: the probability of malfunctioning (Availability) and the consequences of malfunctioning (Severity).

First, this chapter provides some background on current approaches for DSPL evaluation. Then, we have developed a Smart Hotel case study to evaluate the availability and severity of the run-time reconfigurations, following the guidelines for case study research by Runeson and Höst [125]. The Smart Hotel reconfigures its services according to changes in the surrounding context. A hotel room changes its features depending on users' activities to make their stay as pleasant as possible. Overall, the case study comprises eight scenarios and eighteen reconfigurations among these scenarios. The run-time reconfiguration among the different scenarios is the main unit of analysis that we address in this case study. This case study was deployed in a scale environment with real devices to represent the Smart Hotel with human subjects participating in the evaluation (university students performing their senior thesis).

Second, we identify and address two major challenges with the involvement of human subjects in the evaluation. On the one hand, reconfigurations are triggered by context events, many of which are difficult to be reproduced in practice (e.g., a fire). To address this challenge, we have developed a technique that is based on RFID-enabled cards to easily specify the current context. On the other hand, when reconfigurations are performed, some of the effects are easily perceived (e.g., an alarm is triggered) while others are not (e.g., some sensors are deactivated). Thus, we consider that the direct observation of the physical devices is not enough for

evaluating the run-time reconfigurations. To address this challenge, we provided participants with a configuration viewer tool which helps them to understand and evaluate the effects of the reconfigurations.

Furthermore, we also keep track of the experimentation by means of traces of the reconfigurations. These traces gave us insights into the reconfigurations performed by the participants, which contributed to a better understanding of the participant actions, and enabled us to achieve more elaborated conclusions from the experimentation.

The evaluation of the case study reveals positive results regarding both Availability and Severity. However, participant feedback highlights issues with the recovery from a failed reconfiguration or a reconfiguration that is triggered by mistake. To address these issues, we discuss some guidelines learned in the case study. Finally, we conclude that the DSPL achieve satisfactory results with regard to reliability-based risk; nevertheless, DSPL engineers must provide users with more control over the reconfigurations or they will not be comfortable with DSPLs.

8.2 Background on DSPL evaluation

Since DSPL architectures are retargeted to different configurations at run-time, they could benefit from current approaches for adaptive architecture evaluation. Specifically, Yacoub and Ammar [169] proposed a method for reliability risk assessment at the architecture level. This method is based on component-based systems in which implementation entities explicitly invoke each other. Liu et al. [123] also proposed a method for evaluating reliability by means of fault tolerance and fault prevention. They identified architectural design patterns to build an adaptive architecture that is capable of preventing or recovering from failures. Although, these methods do not address run-time reconfigurations that are driven by variability specifications such as Feature Models, they provide techniques (such as estimation of availability and severity) that can be applied in the context of DSPL evaluation.

For SPL evaluation, several approaches have produced results in connection to quality properties such as reliability. For example: the extended goal-based model

[170], the F-SIG Feature-softgoal interdependency graph [171], the Benavides et al. [21] approach, Zhang et al. [172] Bayesian Belief Network (BBN). There are also other methods that are not based on Feature Models: COVAMOF (ConIPF Variability Modelling Framework) [173] and Quality Requirements of a Software Family (QRF) method [174]. Most of these approaches usually remain at the Domain Engineering phase of SPLs only, they do not address run-time reconfigurations as our work does. Therefore, these approaches are not suitable for DSPL evaluation.

Other approaches address reliability evaluation of SPL products at run-time. The RAP approach [175] defines how the reliability requirements should be mapped to the architecture and how the architecture should be analyzed in order to validate whether or not the requirements are met. Etxeberria et al. [176] also take into account reliability at run-time and present a generic approach that can be combined with existing architecture evaluation methods such as PASA [177] or SALUTA [178]. However, since these approaches are oriented to *static products* only, they have to be extended to address the evaluation of *reconfigurable products*, which are the target of DSPLs.

Next sections show the case study that we propose for DSPL reconfigurations, and the challenges that we have identified and addressed to evaluate reconfigurable products.

8.3 The Smart Hotel Case Study

This section introduces the case study of a smart hotel, which reconfigures its services and devices according to changes in the surrounding context. The smart hotel was chosen as the reconfiguration-based case study for two main reasons: first, its nature as a shared environment in which different users use the same room over time. The clients each have their own preferences for the room, which should be adjusted to improve the quality of their stay; secondly, the preferences of the clients change depending on the activity performed (e.g., the clients usually have different preferences when they are watching a movie than when they are working).

Overall, the smart hotel case study describes the stay of one client in different

scenarios. This includes the check-in process and the way the room interacts with the client and changes its features depending on the clients activities in order to make the stay as pleasant as possible. To give an idea of the dimensions of the case study, we present the following metrics:

According to the Feature Modelling technique, the Smart Hotel presents **thirty nine Features**. Some examples of these features are the Temperature Control feature, which offers a heating and cooling system; the Device Synchronization feature which synchronizes the devices that the user can have (e.g., laptop, mp3 player, or PDA) or the Security feature, which secures the room when the user is absent.

The main concepts of the Smart Hotel DSPL architecture are Services, Devices, and the Communication Channels among them. The Smart Hotel has **thirteen Services, twenty Devices and thirty-five Channels**. For instance, the Multimedia Service can establish communication channels to devices such as PDAs or MP3 players.

In the Smart Hotel, users can perform different activities. Specifically, our case study addresses **eight Scenarios**. These scenarios are: Check-in, Entering the Room, Working, Watching a Movie, Sleeping, Leaving the Room, House-keeping and Check-out.

Appendix A describes in detail the Smart Hotel case study. This description comprises all the scenarios that conform the case study by means of the feature modelling technique, the PervML language and reconfiguration tables between the scenarios.

8.3.1 Reconfiguration Scenarios of the Smart Hotel

This section provides a brief description of all the scenarios that make up the Smart Hotel case study. These scenarios cover possible situations that can occur in the smart room of a hotel. The descriptions also indicate the goal of each scenario from the point of view of reconfiguration.

Check-In. When the user registers (online from the internet or at the hotel's reception desk), he is provided with a wizard that makes a few questions to set up the room according to his preferences.

Goal: To reconfigure the room according to the preferences of each user.

Entering the room. When the user enters the room, the smart room detects all the devices that the user is traveling with.

Goal: To integrate the functionality of the user's devices with the room services.

Activity. The room reconfigures itself according to the activities that the user performs in it. The activities can be working, watching a movie or sleeping.

Goal: To reconfigure the room services according to the specific activity that the user is performing at any given moment.

Leaving the room. When the user leaves, the room is reconfigured to disable the services that are no longer needed. Because no one is in the room, it is reconfigured to save energy. The room takes into account when the user has planned to come back (agenda) so that the room is the conditions preferred by the user (illumination and temperature).

Goal: To save energy while there are no users in the room without disturbing them when they come back.

Housekeeping. The room is reconfigured to guarantee the user's privacy when the cleaning service is working in the room. All displays where personal information of the client can be obtained (e.g., TV) are disabled to guarantee privacy.

Goal: To guarantee the user's privacy when the user is not in the room but the hotel staff is.

Check-Out. Finally, when the user finishes the stay in the room, the smart room stops being personalized for that user and its services are reconfigured in order to save energy.

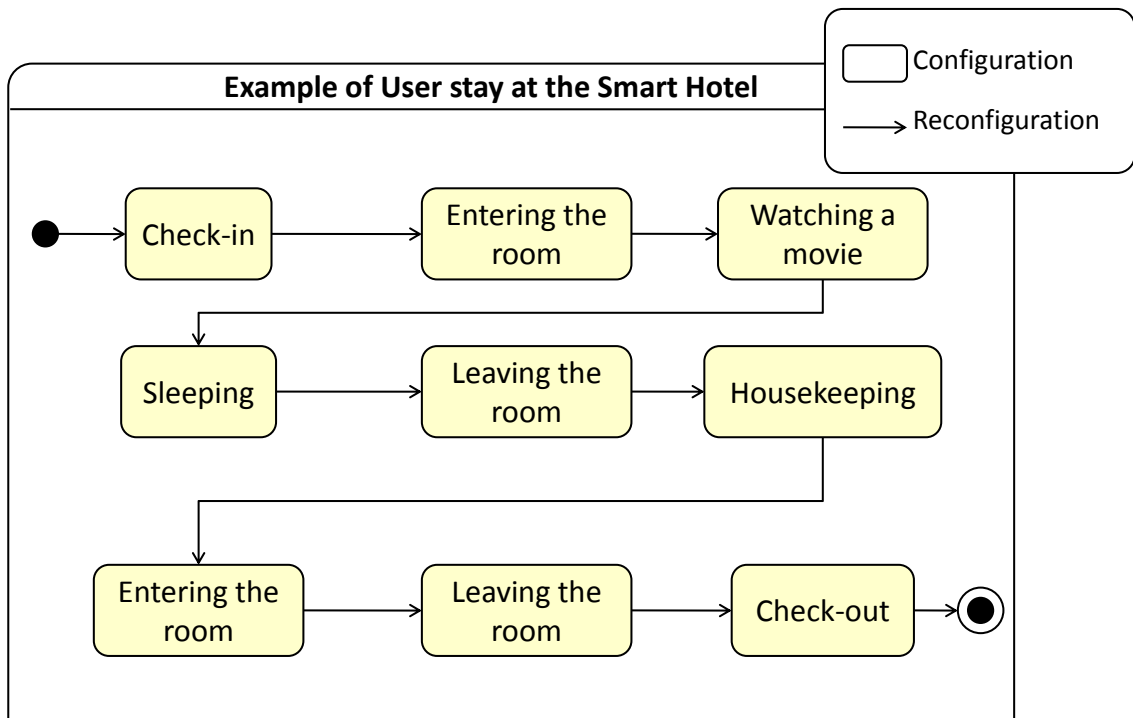


Figure 8.2: Reconfigurations among Scenarios.

Goal: To reconfigure the room to energy-saver mode for the periods when there is no user using it.

By combining the above introduced scenarios introduced above in different ways, we can describe a user's stay at the Smart Hotel. For example, the user checks in the hotel (Check-in scenario) at the reception desk. When he receives his room card, he can immediately enter his room. When he enters the room (Entering the room scenario), he has some free time and he decides to watch a movie selecting one from the hotel's pay-per-view service (Watching a movie scenario). Since it is late, after watching the movie, the user decides to go to sleep (Sleeping scenario). The next morning, the system wakes him up at the time that he has scheduled. The user leaves the room (Leaving the room scenario). During the user's absence, the hotel's cleaning service performs the room's maintenance (Housekeeping scenario). When the user comes back (Entering the room scenario), he has to pack everything to return home. When everything is prepared, he leaves the room (Leaving the room scenario) and then checks out at the hotel's reception desk (Check-out scenario).

Figure 8.2 uses the notation of the state machines to show the above example of user's stay at the Smart Hotel. States represent the scenarios and the transitions indicate valid reconfigurations between scenarios. Appendix A provides details about how all the scenarios are connected with each other. For example, once the user has left the room (Leaving the room scenario), the room can be reconfigured to the following scenarios: Housekeeping, Entering the room, or Check-out scenario.

8.4 Evaluation Logistics of the Case Study

In this case study, we are concerned with reliability-based risk of the run-time reconfigurations. This reliability-based risk depends on the probability that the software product will fail in the operational environment and the adversity of that failure. For the purpose of this work, we have adopted the definition in [169], which defines risk as a combination of two factors: probability of malfunctioning (Availability) and the consequences of malfunctioning (Severity). The probability of failure depends on the probability of the existence of a fault combined with the possibility of exercising that fault. Whereas a fault is a feature of a system that precludes it from operating according to its specification, a failure occurs if the actual output of the system for some input differs from the expected output [169].

It is difficult to find exact estimates for the probability of failure of individual components in the system. In this paper, we adopt the severity classification used in [169, 123] (see Table 8.1). We use a coarse-grained scale, defined as high (H), middle (M), and low (L). We did not adopt an ordinal scale (e.g., 1 to 5) because the values do not truly represent the differences between scales in ratio or distance. In fact, the differences in their values only give indications of their relative rankings. If needed, the scaling definition can be refined later to be more fine-grained or an ordinal scale can be used.

8.4.1 Participants and Training

The participants were 5th-year computer engineering students at the Technical University of Valencia, Spain. Specifically, these students were performing their master

Attribute	High	Middle	Low
Availability	No single point failure	Only one single point of failure	The number of single points of failures > 1
Failure Severity	(aka critical) A failure may cause major system damage or loss of production.	(aka margin) A failure may cause minor system damage, delay, or minor loss of production.	(aka minor) A failure may not cause system damage but will result in unscheduled maintenance or repair.

Table 8.1: Scale Definition of Reliability Metrics.

degree thesis under the supervision of the author of this work. In order to motivate the participants, the experimental tasks were part of their master degree thesis tasks. However, the participants were explicitly not advised that the assessment tasks were part of a formal experiment in order to avoid any spurious effect as a result of the participants being aware of being studied (i.e., avoiding the “Good Subject” effect [179]).

For training purposes, there were two lectures (2 h each) covering the main concepts of a DSPL and introducing the case study. The participants were provided with support materials at the beginning of the experiment, which included specific information for each reconfiguration scenario. They also received training on the use of MoRE [180], the Model-based Reconfiguration Engine of the DSPL that supports the case study. MoRE was a fundamental part of the senior thesis of these students.

8.4.2 Challenges to involve Human participants in DSPL Evaluation

Two major challenges were identified and addressed with the involvement of human subjects in the DSPL evaluation. DSPL reconfigurations are triggered by context events, many of which are difficult to reproduce in practice (e.g., a fire).

To successfully evaluate DSPLs, we must **enable participants to trigger those reconfigurations that are relevant for the experimentation**, not only those reconfigurations that can be easily triggered.

When reconfigurations are performed, some of the effects can be easily perceived (e.g., an alarm is triggered) while others are not (e.g., some sensors are deactivated). To successfully evaluate DSPLs, we must **enable participants to understand and evaluate the effects of reconfigurations**. If participants misunderstand reconfiguration effects, they will not be able to apply the classifications scales of Availability and Severity.

Enabling Participants to Trigger Reconfigurations

Reconfigurations in the case study are triggered by different environmental conditions. When participants are experimenting with the reconfiguration scenarios, they should be able to reproduce these situations in order to validate the system reaction. Since many context events are difficult to reproduce in practice (e.g., simultaneous events that occur in different rooms), simulating them is a must.

The control of context events is essential for the evaluation of DSPLs, since context changes are the events that drive the reconfiguration of the DSPL. Mechanisms should be provided to users to allow them to easily change the current context of the system. In this way, users can move from one configuration to another configuration by applying context changes.

In order to provide an intuitive representation of context events that users could manipulate easily, we provided them with cards that depicted these events. The use of the card metaphor was chosen since it is a familiar concept for most people [181].

Each *context card* represents a context event (such as “fire in the room”). During evaluation sessions, the users were given a deck of context cards. The deck included the events that could affect the particular DSPL being evaluated. The users could then make use of the context cards as the building blocks for triggering the reconfiguration of the DSPL.

The design of the context cards was driven by the elements defined in the Smart Hotel ontology. Each card involved a specific instantiation of a class from the ontol-

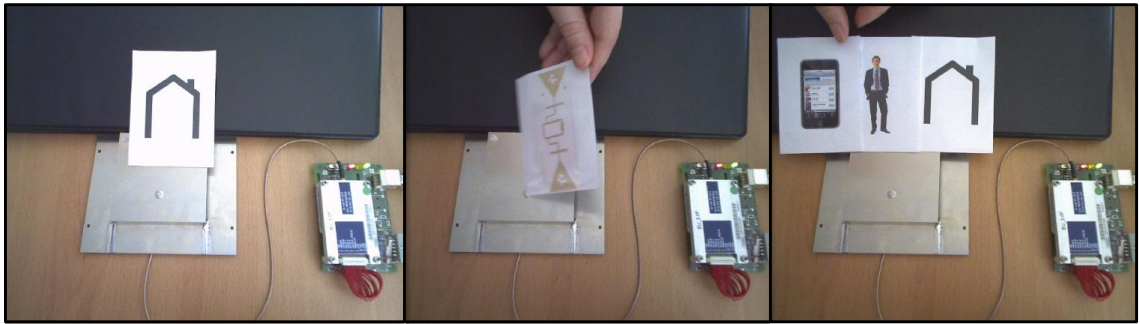


Figure 8.3: Context Cards for triggering DSPL reconfigurations.

ogy. The information provided in the card included the type element and, optionally, some relevant attributes regarding its particular instantiation (such as the location where the event takes place). When the cards were designed, we tried to avoid including too much information. Thus, the users could easily recognize the different cards at a glance (see Fig. 8.3, right).

In order to automate the evaluation process, the Context Cards were enhanced with RFID tags (see Fig. 8.3, left). When a card is placed on the table it is automatically detected by an RFID antenna, and the context ontology is updated accordingly. In this way, the cards can be easily manipulated as if it was part of a card game. Furthermore, they are also closely integrated with the DSPL reconfiguration engine (MoRE). That is, setting a context card close to the RFID antenna triggered the different reconfigurations by means of MoRE.

During the evaluation, the users could add and remove multiple cards from the table in order to define a specific context. The reconfiguration engine reconfigured the DSPL to fit the new context as it changed. Thus, the users could observe how the DSPL was reconfigured as they changed the context events.

The use of context cards enables users to evaluate the reaction of the system in different combinations of context events. Furthermore, putting users in control of the context definition provides valuable feedback. During our evaluation sessions, the users suggested new context cards and specific reconfigurations for certain context combinations that had not been previously considered by designers. Some new context cards were designed to group different events on a single card. Thus, a single card could represent the instantiation of several elements of the Smart Hotel

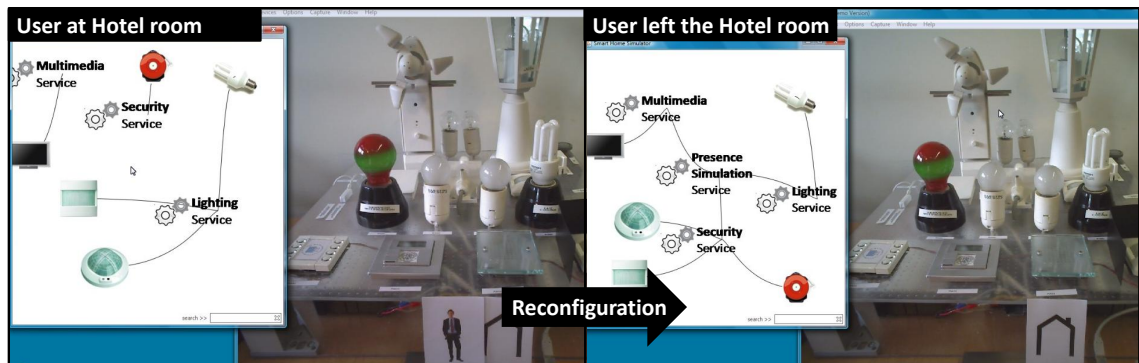


Figure 8.4: Visualizing reconfiguration effects by means of the Configuration Viewer.

ontology. This simplifies the activation of multiple conditions for users.

Enabling Participants to Evaluate the Reconfigurations

According to Dey in [182], one of the biggest challenges to the usability of context-aware applications (as is the case of a DSPL such as [106, 108, 109, 111, 183, 114]) is the difficulty that users have understanding why the applications do what they do. Dey defines the *intelligibility* concept as the support for users in understanding, or developing correct mental models of what a system is doing. This is done by providing explanations of why the system is taking a particular action and supporting users in predicting how the system might respond to a particular input.

Since the DSPLs that we are developing are context-dependant, intelligibility becomes a challenge for their evaluation. When the Smart Hotel is reconfigured, some of the consequences are easily perceivable by users (e.g., an alarm is triggered) while others are not (e.g., some sensors are deactivated). Thus, we considered that the direct observation of the physical devices by the user is not enough for evaluating the DSPL reconfigurations. Mechanisms are required by users to allow them to fully understand the reconfiguration consequences (e.g. changes that are produced in rooms where the user is not present, etc.).

For the evaluation process a Configuration Viewer has been developed to provide users with visual information about the reconfiguration effects in the system. This tool provides a graphical representation of the relevant entities in the Smart Hotel room. These entities include the devices, services, and communication chan-

nels among them. When a context condition is activated, it is also depicted in the Configuration Viewer. Thus, the user can easily perceive that motion sensors are enabled and provide information to the alarm system when the room becomes empty. Without the Configuration Viewer, users cannot be sure whether or not the presence detection has been turned on when they leave the room. As Fig. 8.4 shows, direct observation of the physical devices is not enough to evaluate run-time reconfigurations.

Since we are interested in the evaluation of DSPL reconfigurations, it is not enough to represent the Smart Hotel room in a single state. Therefore, complementary information is provided to the users through our tool to depict what has changed from the previous configurations. By clicking on services or devices, the users get detailed information indicating changes in the configuration (e.g., the motion sensors provide the user with the following message: “motion sensor is no longer in use to control lighting, it is currently in use to control security.”).

This use of the visualization tool enabled users to provide more accurate feedback during the DSPL evaluation since they could determine what has actually changed.

8.4.3 Experiment Operation

In the experimental set-up, a scale environment with real devices was used to represent the Smart Hotel. Therefore, the participants could interact with the same devices that can be found in a real deployment (see Fig. 8.5, top-left). The Configuration Viewer was used during the experiments to keep track of the system evolution. This tool graphically depicts the devices, the services, and the connections among them that are present in the system at any given moment (see Fig. 8.5, bottom-left). Since the reconfigurations are performed as a response to context events, mechanisms are provided for triggering them. We adopted RFID cards to set the Smart Hotel context (see Fig. 8.5, right). Each of the cards symbolized context information such as the presence of users or the occurrence of different events. These cards were combined to insert events in the ontology and to trigger reconfigurations in the Smart Hotel.

During the experiment, the same user interaction with the environment (activat-



Figure 8.5: Experimentation set-up.

ing a presence detector) produced different results according to the current configuration of the system (which depended on the context expressed by the cards). For example, an initial scenario could consist of a room where one inhabitant is present. The cards that defined this scenario are the ones illustrated in Fig. 8.5. In this scenario, the system architecture was organized in such a way that the piped music was available and the presence sensors were used by the lighting service. The user of the prototype could listen to the music and the lights were turned on/off as the user interacted with the sensors. If the card that represented the hotel inhabitant was removed, the sensors were automatically no longer used for the purpose of light control but for security instead. As a consequence, when the user of the prototype interacted with the sensors again, the alarm went off (see this reconfiguration example online¹).

The above description is a small example of the evaluation performance of the reconfigurations introduced by DSPLs. Detailed specifications of the configurations and reconfigurations that make up the case study can be found in <http://www.carloscetina.com/p/hotel.pdf>. There are several videos available about the reconfiguration of our prototype Smart Hotel at <http://www.autonomic-homes.com>.

¹http://www.youtube.com/watch?v=OVtE_RFeEKofmt=22

8.4.4 Data Collection

After each reconfiguration, the participants answered a questionnaire. The questionnaire asked the participant to set the *Availability* and *Failure Severity* for each reconfiguration according to the Scale Definition (see Table 8.1). The participants also indicated the number of context cards that they used to trigger the reconfigurations and whether or not they used the configuration viewer. Finally, the participants answered two questions related to the resulting configurations: “Do you think that the provided reconfiguration is adequate for the context conditions?” and “Do you think that further customization is required to fit your particular needs?”. These two questions required the participants to provide a short explanation.

8.4.5 Keeping Track of the Reconfigurations

In addition to the questionnaires, we also keep track of the experimentation by means of reconfiguration traces. Historically, software engineers have used code-level tracing to capture a running system’s behavior. An alternative is to generate and analyze model-based traces, which contain rich semantic information about the system’s runs at the abstraction level that its design models define.

A model-based trace represents information about the system from a certain viewpoint and omits (or abstracts away) other information. Given a program P and a model M , the model-based execution trace records a run r of P at the abstraction level that M induces.

To support model-based tracing, MoRE stores trace entries each time that a model operation is performed in the context of a reconfiguration (see Figure 8.6). The trace entries range from the conditions which trigger the reconfigurations to the executed reconfiguration plans. Since the reconfiguration are driven by models at run-time, MoRE is able to keep the trace entries at the same abstraction level than the run-time model. That is, both run-time model and trace entries are based on concepts such as features, services or devices.

Given the semantics of the run-time models, an engineer can check if a model-based trace is consistent with regard to a concrete run, and, more generally, if it is

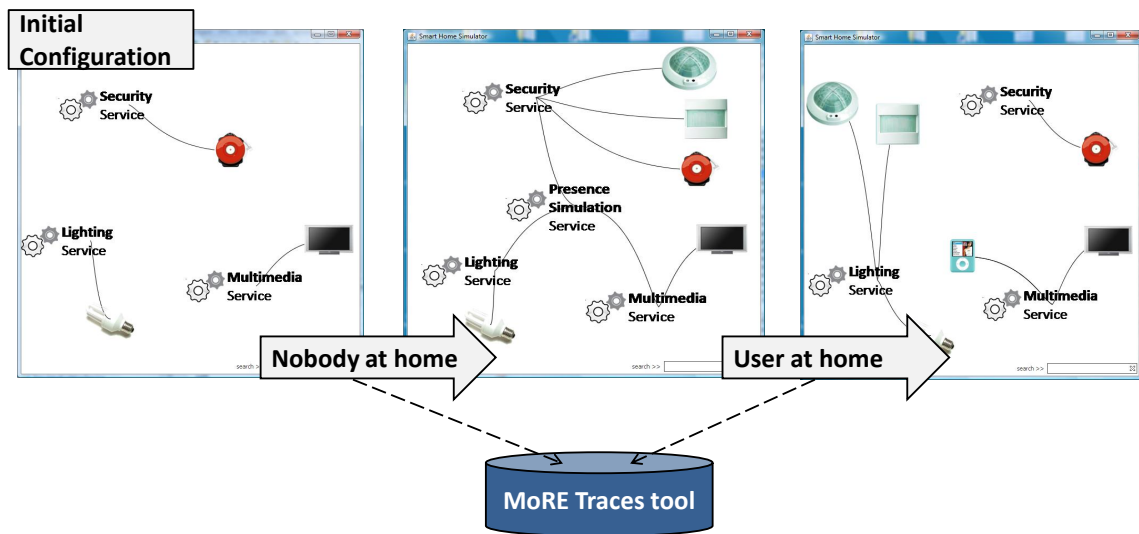


Figure 8.6: Information of the Reconfigurations is Stored as Model-based Traces.

feasible. That is, if a reconfiguration exists from which the trace could have been generated.

The trace entries that MoRE stores can belong to several entry types. We present these entry types and provide examples of their instance creation as follows (see Figure 8.7).

1. **Context Condition.** This entry type provides information about the context conditions that have been fulfilled. Consequently, these conditions are the ones who triggered the reconfigurations. In addition to the context condition information, this entry type also maintains the time stamp of the condition fulfilment.

For example, an instance of this entry type is created when all the users leave the home and the *EmptyHome* condition is fulfilled.

2. **Context Configuration.** This entry type is closely related to context conditions. For each fulfilled condition, a Context Configuration specifies the system changes in terms of features.

For example, an instance of this entry type is created

3. when MoRE processes the Resolution of the *EmptyHome* condition.

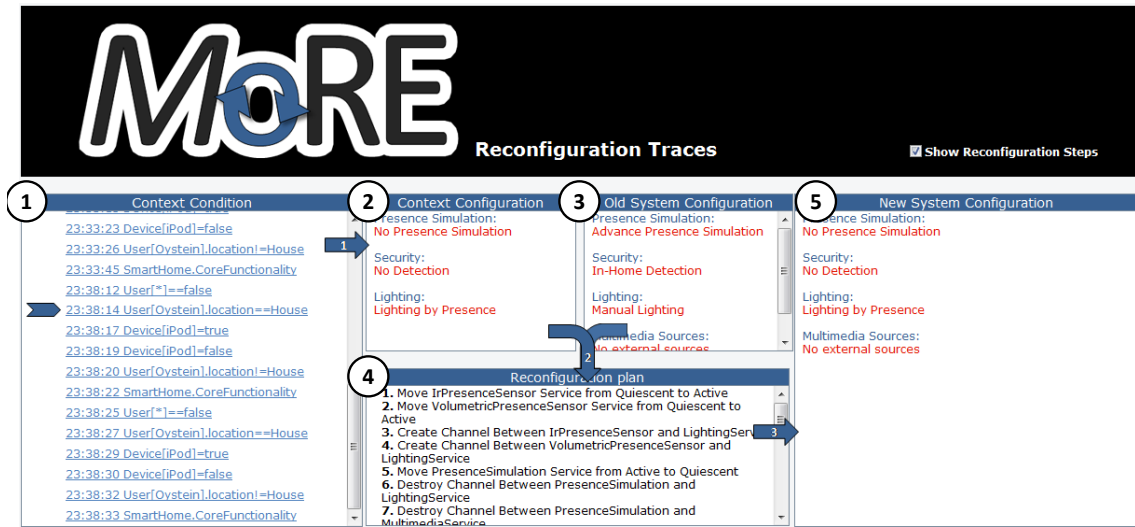


Figure 8.7: Snapshot of MoRE Traces tool.

4. **Old System Configuration.** This entry type provides information about the system configuration before the reconfiguration.

For example, an instance of this entry type is created before the reconfiguration is performed to accommodate the new *EmptyHome* context state.

5. **Reconfiguration plan.** This entry type provides information about the calculated reconfiguration actions to combine both a given Context Configuration and the current System Configuration.

For example, an instance of this entry type is created when MoRE calculates the reconfiguration actions to move the architecture from a *UsersAtHome* configuration to a *EmptyHome* configuration.

6. **New System Configuration.** This entry type provides information about the resulting configuration after the execution of a Reconfiguration Plan.

For example, an instance of this entry type is created after the reconfiguration plan is performed and the *EmptyHome* configuration is reached.

These trace entries provide a way to formally and quantitatively characterize and investigate the concrete reconfiguration the trace was generated from (vertical trace), and also the overall running of the system (horizontal trace).

On the one hand, vertical traces are related to a snapshot of a reconfiguration. They quantitatively reflect the state of a reconfiguration at certain time points in the execution. On the other hand, horizontal metrics are related to an interval of an execution. They are evaluated over a time interval, typically a complete execution or a sequence of connected reconfigurations.

The key characteristic of both vertical and horizontal traces is that they are not mere projections of concrete run-time information onto some limited domain. Rather, they are stateful abstractions, in which trace entries depend on the history and context of the run and the model. The model-based trace not only filters irrelevant information but also adds model-specific information, such as data about entering and exiting reconfigurations that does not appear explicitly in the program code. These model-based traces provides a unique visibility the run-time reconfigurations, and enable us to understand the participants reconfigurations at the same abstraction level that we specified the case study models.

8.5 Evaluation

According to the results of the case study, most reconfigurations (87%) were reported as high *Availability* (see Fig. 8.8). This is mainly due to the fact that the DSPL of the Smart Hotel validates the resulting configuration of each reconfiguration before it is actually performed. If the reconfiguration led to an invalid configuration according to the feature model, then the reconfiguration would not be performed. However, experimentation revealed that even though the configurations were validated, a few of them went wrong in terms of the devices, services, or channels that make up the resulting configuration.

Single points of failure (9% + 4%) were identified mainly on devices and services that were not properly set up in the resulting configuration. In other words, some of these components remained in the old configuration when they were not supposed to, and others changed to a new configuration when they were not supposed to. Several subjects specifically reported that the configuration viewer eased the task of identifying these points of failure. In fact, 92% of the participants made use of the

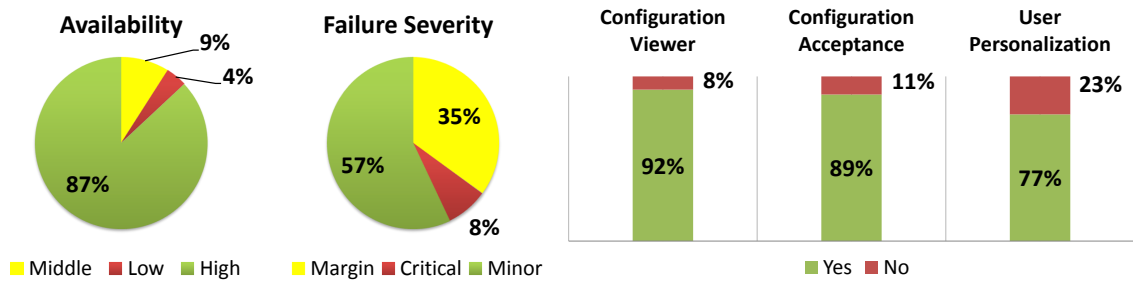


Figure 8.8: Overall results from the Case Study.

configuration viewer. However, they also reported that, in most of the cases, they double-checked the viewer by means of direct interaction with the smart devices and services. It was not until almost all of the scenarios were completed that most subjects began to fully trust the configuration viewer.

Overall, the DSPL reached a high level of *Availability* in most of the case study reconfigurations. However, experimentation revealed that even though DSPLs make use of run-time validation, they are not completely free of reconfiguration failures. To address this issue, we suggest complementing DSPLs with configuration viewers to help users easily detect points of failure.

With regard to the *failure severity*, few failures (8%) were indicated as critical (high severity). These critical failures were mostly related to services that provide inputs to other services. For instance, the *Presence Service* provided inputs to the following services: *Temperature*, *Multimedia*, and *Illumination*. In practice, the lines between producer and consumer services were blurred, and the subjects could not clearly distinguish between them. Hence, when something went wrong it was hard to correctly attribute it to just one specific service. Therefore, the subjects perceived that several services were malfunctioning at the same time. This suggests that services of this kind require more development resources (e.g. testing, quality control, etc.), since they affect the overall perception of the system.

With regard to *context cards* for triggering reconfigurations, Table A.8 indicates the number of cards that were normally used to trigger the case study reconfigurations (minimum and maximum are shown as an indicator). The subjects did not reported any problems related to the understanding of these context cards. In fact, the subjects not only reported new combinations of the current context cards that

should have their own reconfigurations, they also suggested new context cards and reconfigurations for these cards. The context card technique has provided us with interesting insights into the understanding and expectations that users have about reconfigurations. Context cards have not only proven to be a successful technique for setting the context for reconfigurations, but also for capturing reconfiguration requirements. Therefore, we suggest using this technique for both evaluation and requirements elicitation in DSPLs.

With regard to the *configuration acceptance*, we asked the users whether or not they considered the system reaction to be adequate taking into account the defined context events. Acceptance for the reconfiguration scenarios was high (89%). Most of the users considered behaviour provided to be a good response to the context defined with the cards, but they also considered that there was still room for improvement (as illustrated by the user personalization factor).

With regard to the *user personalization factor*, the users were asked whether or not they would modify the system reaction to better fit their needs. Since the specific needs of each user were very diverse (sometimes responding to opposite criteria), we identified the scenarios that could require more fine-grained reconfiguration capabilities. The subjects suggested configuration changes that were important and personally beneficial to them. They transformed configurations from conventional to personal. However, we do not believe that it is economically realistic to build specific features that individually suit participants. Our intent is to focus on commonalities and abstractions that are valid across a set of users, looking for a trade-off between Personalization and Reusability. In fact, the collected data supports that, although participants would modify the case study configurations (77%), most of them thought that the provided reconfiguration is adequate for the context conditions (89%).

8.6 Discussion

Based on our experiences from this case study, we present the lessons that we learned to assist researchers in the context of DSPLs.

8.6.1 Introducing User Confirmations to Reconfigurations

During the evaluation of our DSPL, some subjects reported that they had triggered unintended reconfigurations by mistake. In other words, they mistakenly set up the context for one reconfiguration scenario (i.e. *EnteringTheRoom* - *LeavingTheRoom*), when they really wanted a different reconfiguration scenario (i.e., *EnteringTheRoom* - *Working*). Unintended reconfigurations of this kind were not counted as DSPL failures since they were human mistakes. However, this behaviour raised an interesting point regarding whether or not a reconfiguration should be confirmed before its execution.

After analyzing the unintended reconfigurations performed in our case study, we realized that they can be classified into three different categories. These categories take into account the implications of returning to the source configuration. The three categories are the following:

Round-trip. If there is a reconfiguration that leads directly to the source configuration from the unintended configuration, then we classify the reconfiguration as a round-trip one (see Figure 8.9, left). In our case study, some subjects performed unintended round-trip reconfigurations between *EnteringTheRoom* and *LeavingTheRoom* configurations. For these unintended round-trip reconfigurations, the subjects did not require any special support since they could easily find the way to return to the source configuration. In fact, most of the reconfigurations were not reported as unintended ones in our case study, and those that were reported as unintended did not require support to find the way back. Based on this experience, we do not think that DSPLs should ask for user confirmation before performing a round-trip reconfiguration.

One-way. If there is no reconfiguration that leads directly (or indirectly) to the source configuration from the unintended configuration, then we classify the reconfiguration as a one-way one (see Fig. 8.9, center). In our case study, some of the subjects performed unintended one-way reconfigurations between the *LeavingTheRoom* and *Check-Out* configurations. For these unintended one-way reconfigurations, the subjects always required support since they could

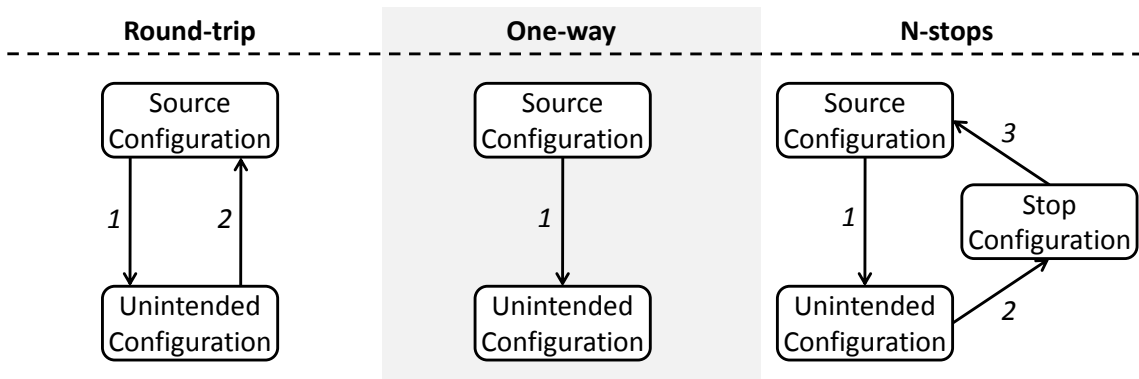


Figure 8.9: Categories for confirmation of reconfigurations.

not find a way back to the source configuration. Based on this experience, we suggest that DSPLs should ask for user confirmation before performing a one-way reconfiguration. This suggestion comes from the fact that once a one-way reconfiguration has been performed, it is not possible to find a way back to the source configuration.

N-stops. If there is a set of reconfigurations that leads to the source configuration from the unintended configuration, then we classify the reconfiguration as a N-stops one (see Fig. 8.9, right). In our case study, some of the subjects performed unintended N-stop reconfigurations between the *EnteringTheRoom* and *Activity* configurations. For these unintended N-stops reconfigurations, almost all the subjects could easily find the way to return to the source configuration. However, a few subjects took a long time to find the way back. Based on this experience, we suggest that DSPLs should ask for user confirmation before performing an N-stops reconfiguration when the number of stops exceeds a certain limit. The purpose of our suggestion is to only require confirmation for critical reconfigurations. We also suggest identifying the acceptable limit of stops by applying Considerate Computing [184] techniques. These techniques take into account the domain particularities of the DSPL in order to determine when the number of reconfiguration stops is not trivial.

Since unintended reconfigurations can occur in DSPLs driven by context events [106, 108, 109, 111, 183, 114] or by user actions [185], we believe that confirmation patterns defined in this study can help DSPLs engineers to mitigate the unintended

reconfigurations. Furthermore, we think that these confirmation patterns are specially relevant for DSPLs driven by context events, since users of these DSPLs usually do not control all the feasible context events and can miss a specific configuration because of it. The confirmation guidelines that came from our case study experience can contribute to avoid this kind of undesired behaviour.

8.6.2 Improving Reconfiguration Feedback

When the subjects of the study perceived the effects of a specific reconfiguration, they sometimes noticed that the result was not the expected one. In those cases, they indicated the presence of a reconfiguration failure, and they also evaluated the severity of the failure. One of the main issues with the evaluation process was related to the termination of the reconfigurations.

Since, each reconfiguration involves changes in different devices, services or communication channels, a delay between the event and the system reaction is introduced. This delay varies from reconfiguration to reconfiguration. Some subjects reported that it was difficult for them to determine whether the reconfiguration process was completed or there were still actions pending. This could lead to misidentifying failure or to miscalculating severity, since a subject could start evaluating a reconfiguration before it was actually finished.

To address this issue, our configuration viewer was enhanced with notification messages that indicated the completion of each reconfiguration. The subjects were provided with feedback regarding the overall process as well as at the service/device level. When a service or device was in the process of reconfiguration, it was depicted as busy (a waiting icon) in the configuration viewer.

Most of the subjects reported that they found this reconfiguration feedback to be very useful not only for failed reconfigurations but also for regular reconfigurations. Therefore, we suggest that DSPLs should provide feedback about the termination of reconfigurations, especially, when reconfigurations involve human users.

8.6.3 Introducing Rollback Capabilities to Reconfigurations

Our case study raised another important concern in connection with DSPL recovery after a failure. Once a reconfiguration failure was identified and evaluated, a few subjects required support to resume the experimentation. They reported problems in performing the next reconfiguration after the failure. In other words, they did not find a simple way to reach another configuration of the case study. Below, we present the main kinds of issues reported and how we think they should be addressed in DSPLs.

Unexpected configurations. After a failure reconfiguration, a few subjects reported that the resulting configuration was not the expected one. In place of the expected configuration (i.e., *WatchingAMovie*), they got another configuration (i.e., *Working*). In most of these cases, the subjects could perform a new reconfiguration in order to reach the expected configuration. However, a few of the cases required several reconfigurations to reach the expected configuration. To address this issue, in DSPLs, we suggest introducing some sort of “undo” operation that returns the system directly to the previous configuration.

This has several implications for the design of DSPLs since some actions have collateral effects that cannot be easily undone (e.g., sending an e-mail). The handling of compensation actions to reverse a reconfiguration should be studied, also the consequences of a rollback need to be explained so that users can be provided information to help them choose among different compensation actions and understand how they relate to their desired goals.

Unknown configurations. After a failure reconfiguration, some subjects reported that they failed to identify the resulting configuration in the Smart Hotel documentation. In other words, the resulting configuration was different from all the documented configurations that made up the case study (see Figure 8.2). The Feature Model of the Smart Hotel defines more configurations than the ones considered in our case study. These *unknown configurations* imply that the subjects could not identify the set of reconfigurations that led to the expected configuration. Therefore, they needed support to continue the

experimentation. To address this issue, we strongly suggest an “undo” operation that returns the system directly to the previous configuration. Note that for *Unknown configurations*, we think that the “undo” operation should be mandatory. However, for *Unexpected configurations*, we think that the “undo” operation should be optional since users have an alternative to achieve the expected configuration.

The DSPL that supports this case study makes use of Feature Models at run-time to determine how to perform the reconfigurations. According to a recent discussion on DSPL architectures [186], other DSPL approaches make use of different techniques to perform reconfigurations (i.e., QoS properties or UML profiles). Although the details are different, these DSPLs are based on variability specifications, and their reconfiguration can also lead to *Unexpected configurations* or *Unknown configurations*. Even though these DSPLs could achieve an expected configuration from any given *Unexpected* or *Unknown* configuration, our experience suggests that introducing an “undo reconfiguration” operation is simpler and more practical from the viewpoint of the DSPL user.

8.7 Conclusions

With more and more devices being added to our surroundings, simplicity becomes greatly appreciated by users. Dynamic Software Product Lines (DSPL) encompasses systems that are capable of modifying their own behavior with respect to changes in their operating environment by using run-time reconfigurations. However, failures in these reconfigurations directly impact the user experience since the reconfigurations are performed when the system is already under user control. This is in contrast to SPLs where all the configurations are performed before delivering the system to the users.

Given the importance of run-time reconfigurations in DSPLs, we have evaluated the reliability-based risk of these reconfigurations, specifically, the probability of malfunctioning (Availability) and the consequences of malfunctioning (Severity). The evaluation has been performed by means of the Smart Hotel case study which

was deployed with real devices with the participation of human subjects.

Furthermore, we successfully identified and addressed two challenges associated with the involvement of human subjects in DSPL evaluation. On the one hand, DSPL reconfigurations are triggered by context events many of which are difficult to reproduce in practice. To evaluate DSPLs, we successfully applied a technique based on Context Cards to enable participants to trigger reconfigurations. On the other hand, when reconfigurations are performed, some of the effects are easily perceived (e.g., an alarm is triggered) while others are not (e.g., some sensors are deactivated). For this problem, we successfully applied a technique to enable participants to understand and evaluate the effects of reconfigurations. If participants misunderstand the reconfiguration effects, they will not be able to apply the classification scales of Availability and Severity. We believe that these techniques can also contribute to the evaluation of more quality properties in the context of DSPLs.

The evaluation of the case study reveals positive results regarding both Availability and Severity. We hope that these positive results encourage researchers and practitioners to apply DSPL to other promising areas of research such as mobile devices or automotive systems. However, the participant feedback in this study highlights issues with recovery from a failed reconfiguration or a reconfiguration triggered by mistake. To address these issues, we have provided some guidelines learned in the case study.

Finally, we conclude that the DSPL has achieved satisfactory results regarding reliability-based risk; nevertheless, DSPL engineers must provide users with more control over the reconfigurations or the users will not be comfortable with DSPLs even though they achieve a high level of reliability.

Chapter 9. CONCLUSIONS AND FUTURE WORK

“Though no one can go back and make a brand new start, anyone can start from now and make a brand new ending.”

– Carl Sandburg (1878-1967).

9.1 Overview of the Chapter

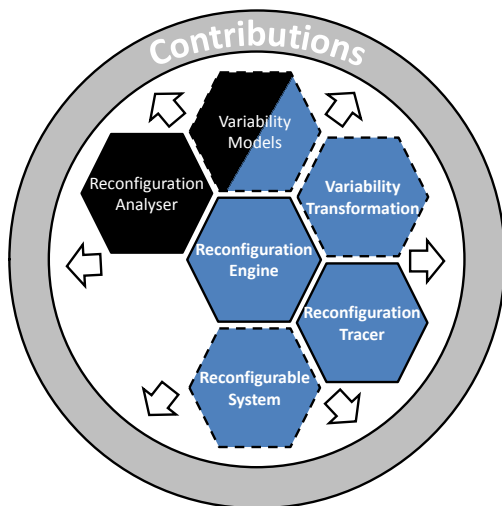


Figure 9.1: Scope of Chapter 9

This thesis has investigated the use of variability models at run-time to achieve autonomous computing. Our research shows the feasibility of achieving autonomous behavior by leveraging variability models at run-time. In this way, the modelling effort made at design time is not only useful for producing the system but also provides a richer semantic base for autonomous behavior during execution. We applied our approach to the smart home domain. This domain is suited for variability modelling techniques because of the high degree of similarities among different systems;

also, autonomous computing capabilities can address some of the domain’s limitations such as minimal support for evolution as new technologies emerge or as an application type matures.

Whether in smart homes, mobile devices or automotive systems, end-users require more and more autonomic functionality. We consider that the techniques applied for the Smart Home domain can also be applied to other environments with similar results.

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

9.2 Contributions

The major contribution of this thesis is a **software engineering approach for autonomic computing** which combines the main ideas of Model Driven Development (models as first-order citizens) and Software Product Lines (variability management). This approach provides not only an execution platform but also techniques and tools to support autonomic system engineers from system design to execution. In particular, we have demonstrated that systems can make use of the knowledge captured by variability models as if they were the policies that drive the autonomic behaviour of the system at run-time. This main contribution is complemented with two other contributions.

1. We show **how to design and validate the autonomic behaviour** by means of variability modelling techniques (either Feature Modelling or CVL specifications) and the FaMa framework for variability analysis.
2. We provide **a model-based implementation of the reference model for autonomic control** [2] in order to support the overall approach.

Although, the above contributions push towards a sound and seamless engineering support for autonomic computing. We believe that this thesis also provides remarkable results for both *Models@run-time* [187] and *Dynamic Software Product Line* [188] communities as follows.

- Relevant results for the *Models@run-time* community:

- **The demonstration of the feasibility of keeping the same model representation at run-time that is used at design time:** the XML Metadata Interchange (XMI) standard. In our experiments, we used an XMI model at run-time in order to determine how to query and update it using the widespread tools of the Eclipse Modelling Project. This avoids the definition of technological bridges, because the same technologies used at design-time for manipulating XMI models can be applied at run-time. We have also shown the feasibility of this approach from the point of view of efficiency.
- **A Model-based Reconfiguration Engine (MoRE)** which uses both variability models and variability transformations at run-time to determine how a system should be reconfigured for a target operational context. This engine also provides the mechanisms for modifying the system architecture accordingly. Furthermore, this engine support two main techniques for variability modelling: Feature Modelling and CVL specifications.
- **The realization of the run-time Variability transformation by means of interchangeable strategies.** These strategies enable engineers to use the most suitable strategy for each concern, because these strategies cover specific extra-functional requirements such as performance or support to reconfiguration debugging at run-time. We compared these strategies from the viewpoint of the extra-functional properties, and we also gave recommendations to use the most suitable strategy for different concerns of run-time reconfiguration.
- **A testing approach to validate the implementation of run-time strategies for variability transformation.** This approach systematizes the detection of differences between a new strategy implementation and a reliable one. Furthermore, the approach also provides information about the validation of a strategy when we do not have a reliable strategy yet. We also provide a tool to automate the whole testing process.

- Relevant results for the *Dynamic Software Product Line* community:
 - **The identification and solution of two challenges** associated with the involvement of human subjects **in DSPL evaluation**: to (1) trigger run-time reconfigurations and to (2) understand the effects of the reconfigurations. These techniques can be applied not only to reliability-based risk but also to other quality properties that require the execution of reconfigurations by human users, for instance, usability or security.
 - **A case study** that is representative of real problems (Smart Hotel), which has been specifically developed to exercise reconfigurations of DSPLs, and which has proven to be well-understood by users in experimentation. Since the design of case studies is recognized as a difficult step during the development of experimentation [189], we believe that the Smart Hotel case study can be applied to more empirical research in the context of DSPL. Detailed documentation about this case study is publicly available online at <http://www.carloscetina.com/papers/smart-hotel.pdf>.
 - **The experimentation results**, which reveal the maturity of run-time reconfigurations with regard to both Availability and Severity. These results can encourage researchers and practitioners to apply DSPL to other promising domains.
 - **The identification of key issues for user acceptance of DSPLs**: to (1) recover from a failed reconfiguration, and to (2) recover from a reconfiguration triggered by mistake.
 - Specific **guidelines for addressing the identified issues** of recovery from a failed reconfiguration or a reconfiguration triggered by mistake.

We hope that these contributions encourage researchers and practitioners to apply *reconfigurations through variability models at run-time* to other promising areas of research such as mobile devices, automotive systems or resilience system.

9.3 Research Visits

The aim of this work was to be open, influenced and enriched by distinct research streams, works, visions and schools. Thus, along this work three research visits were accomplished.

1. **Destination.** Object orientation, Modelling and Language Group (OMS), University of Oslo and Sintef, Norway.

Host. Prof. Dr. Øystein Haugen.

Duration. From October to December 2008.

Relevance for the thesis. The work was in connection to the Model-driven development of highly configurable embedded Software-intensive Systems (MoSiS) ITEA project. The relevance of the MoSis project for this thesis comes from the fact that the MoSis project also addresses the applicability of variability modelling and reconfigurable architectures for run-time adaptability.

Results. Throughout the stay, we discussed different strategies to realize the variability transformation at run-time. These strategies enhanced MoRE to address concerns such as performance or debugging.

2. **Destination.** Applied Software Engineering Research Group (ISA), University of Seville, Spain.

Host. Prof. Dr. Antonio Ruiz-Cortés.

Duration. April 2009.

Relevance for the thesis. The work was in connection to the Framework for the Automated Analysis of Feature Models (FaMa Tool Suite). This framework enables to determine if a system configuration is valid (according to variability constraints), and it can also provide explanations about invalid configurations.

Results. Throughout the stay, we discussed different approaches to applied FaMa at run-time by means of the OSGi framework. We also addressed how to integrate both FaMa and MoRE to improve run-time reconfigurations.

3. **Destination.** Object orientation, Modelling and Language Group (OMS),

University of Oslo and Sintef, Norway.

Host. Prof. Dr. Øystein Haugen.

Duration. From September to December 2009.

Relevance for the thesis. The work was in connection to the Model-driven development of highly configurable embedded Software-intensive Systems (MoSiS) ITEA project.

Results. Throughout the stay, we discussed how to address the design of reconfigurations driven by variability models. Specifically, we were interested on how to avoid MoRE leading a system to invalid configurations at run-time.

These visits fostered discussion and eventually imposed new perspectives on this work that otherwise would not be reached.

9.4 Assessment and Future Work

A desirable aspect of any research is that in addition to providing solutions to initial issues or questions, it should identify new areas of research that would allow researchers to eventually produce more useful knowledge and progress. In this section we identify many research activities are currently underway, and further research is ongoing in different and complementary directions.

9.4.1 Enabling End-user participation in the Design of Reconfigurable Systems

As stated by Christopher Lueg [190], technology developers make assumptions about which aspects of human activities and their physical and social environment are important in future usage situations. In a similar way, run-time reconfigurations may involve assumptions about the desirable functionality of end-users. Conversely, end-users are the ones who best know their activities and their functionality expectations. Hence, we plan to involve end-users in the design of reconfigurations in order to minimize the mismatch between user expectations and system behavior.

Specifically, we are working on a design method for reconfigurable systems 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 plan to complement this method with a specification technique so that both end-users and designers can configure the systems in terms of features. In this method, designers are in charge of defining the functionality blocks in which the system is based, and then end-users determine how these functionality blocks can be combined according to their preferences and needs.

9.4.2 Enhancing Run-time Reconfigurations to Take into Account End-user Preferences

Once the system has been deployed, it must improve everyday life activities without losing user acceptance of the system [191]. Therefore, End-user needs should be taken into account both before and after the system is deployed to keep users from feeling a lose of control [8, 9]. We believe that users need to feel under control although an autonomic sytem makes its own decisions. Therefore, user preferences must be taken into account.

To address this issue, we plan to extend the reconfiguration behaviour to change system configuration while user preferences are taken into account. To perform this extended reconfiguration, we will focus on covering the average demand of the system users rather than the preferences of specific individuals.

Whenever new users appear, leave or their preferences change, the autonomic system must analyse its state and determine if there exist another configuration that satisfies most of the user preferences. Our new target objective for the autonomic system is reconfiguring its architecture to maximise the fulfilment of user preferences.

Sometimes, every user preference may be fulfilled at the same time; other times some user preferences could not be partially or completely satisfied as they collide with other user preferences. Finding an optimal configuration is a hard problem and may take time to be solved in some situations. However, in other situations, it may be important to give the fastest response as possible. But fast is frequently incompatible with best solution. Therefore, we plan to limit search in time so not

the best but a good configuration is obtained. It is still possible to keep searching for a better or the best solution in background so later reconfigurations may arise whenever they are found.

9.4.3 Providing Metrics to Quantify System Reconfiguration Capabilities

As for any other software engineering approach, it is furthermore a key concern to answer the question what the measurable benefits of using dynamic adaptation are. In our context this means that it is indispensable to come to a possibility to measure the impact reconfigurations for a particular variability specification. Otherwise it is not possible to evaluate and to compare different variability specifications with reconfiguration purposes. Neither it is possible to evaluate the chosen variability specification and thus to control and to steer the development process.

To address this issue, we plan to define a set of metrics to evaluate the reconfiguration range of variability specifications. Specifically, we target metrics which can be calculated once the designers have defined the variability specification. In particular, we are interested on enabling designers to identify weak points in the variability specification (from the viewpoint of reconfigurations) and on providing an overall estimation of the system adaptation range.

9.4.4 Guarantying Quality Properties on Run-time Reconfigurations

In addition, the role of models at design time can be extensively exploited for the purpose of validation and verification. Since the Variability Models, which determine the autonomic behaviour, are available at design time, it is possible to conduct a thorough analysis of the specifications for the purpose of guarantee quality properties.

Specifically, we plan to extend the step of reconfiguration analysis of our approach in order to guarantee specifications free of (1) Unsafe Reconfigurations and (2) Unsafely reachable configuration. Unsafe Reconfigurations lead to an invalid possibility

from a valid one resulting in an inconsistent system state. Unsafely reachable configurations are valid configurations that can be reached through invalid ones only. We believe that dealing with these properties is essential for reliable systems as a next step in obtaining autonomic systems that fulfill many of the user's needs out-of-the box.

9.4.5 Addressing other Application Domains

Whether for smart homes, mobile devices, or decision support systems, users require more autonomic functionality. We believe the techniques we have applied to the smart home domain can achieve similar results in other domains such as Service Oriented Architecture or Method Engineering as follows.

- The vision of **Service Oriented Architecture** (SOA) promotes an ecosystem of services where there are alternative providers for the services offering different quality levels and prices.

However, this is a dynamic ecosystem where service offers appear and disappear. For clients this means that they have to dynamically select and bind to suitable providers. For providers it means that they have to provide an attractive offer and to serve a varying set of clients with varying needs.

We believe that this work can play a significant role towards the implementation of self-management properties in order to manage Service Level Agreements and to reconcile the client-provider negotiation of SOA ecosystems.

- **Method Engineering** is the engineering discipline to design, construct and adapt methods, techniques and tools for the development of information systems.

However, the focus has been on the efficient derivation of a customized method to meet the requirements of a particular project, that, once created, remains static throughout their lifetime.

We believe that the ideas presented in this work can enable method engineers to also face dynamic concerns such as *human resources fluctuation*, or *tasks*

reschedule in order to achieve Dynamic Method Engineering.

9.5 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.

A) International Journal papers indexed in the First Quartile of JCR by Thomson Reuters

1. **Carlos Cetina**, Pau Giner, Joan Fons, & Vicente Pelechano. Autonomic computing through reuse of variability models at run-time. *IEEE Computer*. 2009.

Impact Factor 2008: 2,093. Category 2008: Computer Science, Software Engineering 16/86.

2. Pau Giner, **Carlos Cetina**, Joan Fons, & Vicente Pelechano. Developing support for mobile business processes in the internet of things. *IEEE Pervasive Computing*. 2010.

Impact Factor 2008: 2.615. Category 2008: Computer Science, Information System 12/99.

B) International Journal papers not indexed in the JCR

3. Pau Giner, **Carlos Cetina**, Joan Fons and Vicente Pelechano. Orchestrating your Surroundings. *ERCIM News*. 2009.

4. Javier Muñoz, Estefanía Serral, **Carlos Cetina** and Vicente Pelechano. Applying a Model-Driven Method to the Development of a Pervasive Meeting Room. *ERCIM News*. 2006.

C) Book Chapters in International Books, excluding Conference Proceedings

5. **Carlos Cetina**, Joan Fons & Vicente Pelechano. The Adoption of Software Product Lines to Develop Autonomic Pervasive Systems. Book chapter of

Applied Software Product Line Engineering. Taylor and Francis (Edited by Vijay Sugumaran, Kyo Kang and Sooyong Park). 2009.

D) (Highly-ranked) International Conference Papers Indexed in ISI by Thomson Reuters and Acceptance Rates <30%

6. **Carlos Cetina**, Øystein Haugen, Xiaorui Zhang, Franck Fleurey & Vicente Pelechano. Strategies for Variability Transformation at Run-time. 13th International Software Product Lines Conference (SPLC). San Francisco, California. 2009.

Most prestigious Conference in SPL Field¹. Conference Acceptance Ratio: 29%.

7. **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.

Most prestigious Conference in SPL Field¹. Conference Acceptance Ratio: 28%.

E) International Conference Papers Indexed in the First Tier of CORE Ranking

8. Pau Giner, **Carlos Cetina**, Joan Fons, Vicente Pelechano. Presto: A plug-gable platform for supporting user participation in Smart Workflows. The Sixth Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous). Toronto, Canada. 2009. CORE 2009: A. Conference Acceptance Ratio: 20%.

F) International Conference Papers published by IEEE Computer or Springer

9. **Carlos Cetina**, Pau Giner, Joan Fons & Vicente Pelechano. Using Feature Models for Developing Self-Configuring Smart Homes. The Fifth International Conference on Autonomic and Autonomous Systems (ICAS). Valencia,

¹According to Carnegie Mellon's list of conferences, "SPLC is the most prestigious and leading forum for researchers, practitioners, and educators in the field of Software Product Line Engineering"

Spain. 2009.

Conference Acceptance Ratio: 23%.

10. Pau Giner, **Carlos Cetina**, Joan Fons & Vicente Pelechano. Building Self-adaptive services for Ambient Assisted Living. Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing and Ambient Assisted Living. Salamanca, Spain. 2009.

Conference Acceptance Ratio: 15%.

11. 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.

G) International Workshop Papers published in High-Relevance Forums to this Thesis

12. **Carlos Cetina**, Pablo Trinidad, Vicente Pelechano, Antonio Ruiz-Cortés. Customisation along Lifecycle of Autonomic Homes. 3rd International Workshop on Dynamic Software Product Lines (DSPL). Limerick, Ireland. 2009.

13. **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). Toulouse, France. 2008.

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

15. Francisca Pérez, **Carlos Cetina**, Pedro Valderas, Joan Fons. Towards End-User Development of Smart Homes by means of Variability Engineering. Third International Workshop on Variability Modelling of Software-intensive Systems (VAMOS). Sevilla, Spain. 2009.

16. **Carlos Cetina**, Javier Muñoz, 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.

H) National Conferences (JISBD is the main Spanish conference on Software Engineering)

17. **Carlos Cetina**, Pablo Trinidad, David Benavides, Vicente Pelechano & Antonio Ruiz-Cortés . Moskitt FM and FAMA FW: Taking feature models to the next level. XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD). San Sebastián, Spain. 2009.

18. **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). Gijón, Spain. 2008.

I) International and National Publications related to the PhD Courses

19. **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.

20. Jose Manuel Marquez Vazquez, **Carlos Cetina**, Francisco Velasco, Luis Gonzalez-Abril, Juan Antonio Ortega. Modelado de características para itinerarios formativos adaptativos. X Jornadas de ARCA. Sistemas Cualitativos y Diagnosis, Robótica, Sistemas Domésticos y Computación Ubicua (JARCA). Tenerife, Spain. 2008.

21. Javier Muñoz, Vicente Pelechano, **Carlos Cetina**. Software Engineering for Pervasive Systems. Applying Models, Frameworks and Transformations. IEEE International Conference on Pervasive Services (ICPS). Istanbul, Turkey. 2007.

22. **Carlos Cetina**, Estefania Serral, Javier Muñoz, 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.

23. Estefania Serral, **Carlos Cetina**, Javier Muñoz, Vicente Pelechano. PervGT: Herramienta CASE para la Generación Automática de Sistemas Pervasivos. XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD). Zaragoza Spain. 2007.
24. Muñoz, **Carlos Cetina**, Estefanía Serral, Vicente Pelechano. Framework basado en OSGi para el Desarrollo de Sistemas Pervasivos. Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS). La Plata, Argentina. 2006.
25. Vicente Pelechano, Manoli Albert, Javier Muñoz, **Carlos Cetina**. Building Tools For Model Driven Development. Comparing Microsoft DSL Tools and Eclipse Modelling Plug-Ins. III Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones(DSDM). 2006.
26. Javier Muñoz, Vicente Pelechano, **Carlos Cetina**. Implementing a Pervasive Meetings Room: A Model Driven Approach. International Workshop on Ubiquitous Computing (IWUC). Paphos, Cyprus. 2006.
27. Javier Muñoz, Idoia Ruiz, Vicente Pelechano, **Carlos Cetina**. Un framework para la simulación de sistemas pervasivos. Simposio sobre Computación Ubicua e Inteligencia Ambiental (UCAmI). 2005.

9.6 Senior Theses Codirected

The results presented in this thesis, specially the developed tools, have been put to the test on distinct Senior Theses directed by the author of this thesis. The Senior Theses in which the author of this thesis was involved as codirector are listed below.

1. Servicios Reconfigurables para Hogares Inteligentes. Esteban Saiz Martínez. School of Engineering in Computer Science, Technical University of Valencia. 2010.
2. Configuración de Casas Domóticas con la Tecnología RFID. Alfons Vicente Gomez Ferragud. School of Engineering in Computer Science, Technical University of Valencia. 2010.

3. Computación Autónoma aplicada a Hogares Digitales. David Unió Miralles. School of Engineering in Computer Science, Technical University of Valencia. 2009.
4. Desarrollo de una Arquitectura Software Reconfigurable para Hogares Digitales. Ignacio Climent Romero. Faculty of Computer Science, Technical University of Valencia. 2009.
5. Líneas de Producto Software Dinámicas aplicadas a Hogares Digitales. Salvador Ibiza Molines. Faculty of Computer Science, Technical University of Valencia. 2009.
6. Diseño e Implementación de una Arquitectura para Orquestrar Servicios en Sistemas Pervasivos. María Francisca Perez Perez. Faculty of Computer Science, Technical University of Valencia. 2008.
7. Aplicación de Líneas de Producto a Entornos de Inteligencia Ambiental en la Plataforma Eclipse. Sonia Montagud Gregori. Faculty of Computer Science, Technical University of Valencia. 2007.
8. Diseño, Desarrollo e Implementación del Portal Web del Proyecto de Investigación SEAPS. Daniel Sainz García-Cernuda. School of Engineering in Computer Science, Technical University of Valencia. 2007.

9.7 Seminars

Thanks to the impact of this work in the community, the author was invited to participate in the Dagstuhl Seminar: Software Engineering for Self-Adaptive Systems, Germany, October 2010 (see <http://www.dagstuhl.de/10431/>).

According to the organizers, the goal of the above seminar is to bring together the leading software engineering experts and other distinguished experts from related fields on self-adaptive systems to discuss the fundamental principles, models, methods, techniques, mechanisms, state-of-the-art, and challenges for engineering self-adaptive software systems.

9.8 Final Conclusion

Henry Ford, founder of the car company that bears his name, is widely regarded as the father of assembly-line automation, which he introduced and expanded in his factories producing Model Ts between 1908 and 1913. In his book *My Life and Work* (1922), he stated the following:

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

This means that on mass-production environments such as those involved in cars or houses, production costs must be taken as a major constraint. Reducing production costs comes at the expense of limiting the level of detail in personalization. For example, when buying a car, you can choose the color but only from a limited catalogue.

In our experimentation, we found some scenarios which required a greater level of detail to define the autonomic behaviour since these scenarios deal more directly with user preferences and tastes. However, even though this lack of coverage could be complemented by the development of specific components for the unsupported cases, it does not seem economically realistic to build individual features to suit each user. Our intent is to focus on commonalities and abstractions that are valid across a set of users, looking for a trade-off between personalization and reusability. This trade-off is acceptable in these domains since, in general, the focus is on covering the average demand, not the needs of each individual.

Appendix A. THE SMART HOTEL

CASE STUDY

This appendix presents the case study of a Smart Hotel, which reconfigures its services according to changes in the surrounding context. The choice of the smart hotel as a case study comes from two main reasons. First by its nature of shared environment in which different customers use the same room over time. Each client has their own preferences for the room and it should be adjusted to improve the customer's stay. Secondly, the preferences of a user changes depending on the activity performed. For example, different preferences when you are watching a movie or when you are working.

Overall, the smart hotel case study introduces the stay of Professor John. This includes the process to check-in in the hotel and after that how the room interacts with him and changes its features depending on professor's activities to make his stay as pleasant as possible.

Here are some metrics of the case study to give an idea of its dimensions.

- According to the Feature Modelling technique, the Smart hotel presents 39 Features. Some examples of these features are the Device Dock feature that can work as a device to charge or synchronize all the devices that the Professor can have (For example, Laptop, mp3 player or PDA), the Temperature Control feature that offers a heating and cooling system or the Security feature that secures the room when the professor is out of it.
- The main concepts of the PervML DSL are Services, Devices and the Communication Channels among them. The Smart Hotel is composed by 13 Services, 20 Devices and 35 Channels. For instance, the Multimedia Service can es-

establish communication channels to devices such as laptops, PDAs or MP3 players.

- In the Smart Hotel, users can perform different activities. Specifically our case study addresses 8 Scenarios. These scenarios are: Check-in, Entering the Room, Working, Watching a Movie, Sleeping, Leaving the Room, House Keeping and Check-out.

This appendix is organized as follows. First, we present an overview of how the room reconfigures itself when a user is interacting with it. Second, we provide a brief introduction to all the scenarios that conform the hotels room, and how are these scenarios connected among them. For example, after the check-in scenario it is only possible to go to the entering the room scenario and not to others like watching a movie or sleeping. Next, we present the different room configurations by means of a Feature Model. The Smart Hotel architecture is presented using the PervML language to describe the services, devices and communications channels in the room. Finally, we provide a full description of each scenario about how the room reconfigures its services according to a particular context. Each scenario is specified by a Feature Model showing its current configuration and a PervML model with the services and devices for the particular context. A reconfiguration table is also shown to indicate how the room reconfigures itself when there is a change of scenario.

A.1 Overview of the Case Study

In this section we describe with a detailed example how all the scenarios explained throughout this chapter come into operation. Professor John helps us describing how the room reconfigures itself according to his preferences and actions.

Professor John is going to a conference on a different country for a few days. Professor John has a tight schedule during his trip and he needs to be very strict with his appointments. The moment he arrives at the hotel, he receives a card that is the key to enter to his room. He uses the card on the door's card reader to identify himself. If everything is correct the room's door will open automatically allowing

him to get into the room. When he gets inside, the first thing he sees is a big screen welcoming him and asking to answer a few questions. Those questions are to know a bit about professor's preferences. He has his entire schedule in his laptop, so he has the possibility to synchronize that information with the room's system. This will help the room to know all the appointments the professor has to attend and try to make him remember for not being late to any of them. He also can connect his cell phone to the room's main system via Bluetooth and attend the calls without having to hold it.

Professor just arrived from a long trip and is a bit tired. He can connect his own devices to the room control system to listen to his own music or watch a movie while having massages from the sofa in the room. The lights in the room get softer so he can feel much better.

After the relaxing moment, professor decides to prepare some necessary work for the next day. The coffee machine is ready for whenever the professor decides to drink and keep working. The room will take in account the time when professor uses it for possible next times.

When professor wants to sleep, the lights in the room will turn off gradually. During the night, professor needs to go to the toilet. The room system will detect that and the lights will turn on with enough illumination so the professor can clearly see the way without hurting his view.

It's time to wake up and the room simulates a sunrise so the transition to get awake is as smooth as possible. The coffee machine will also be ready so the professor loses as less time as possible. The entire schedule for that day will be displayed on the room's main screen so professor doesn't miss anything.

When professor is out, the room will try to save energy until the moment he comes back. It will also hide the entire professor's personal information to keep his privacy when the rooms' hotel service goes to clean.

It's time for the professor to go back to his country so the moment when he checks out, the main screen will inform that all the shared information will be deleted, hoping he had a pleasant stay and inviting him to come back again in the future.

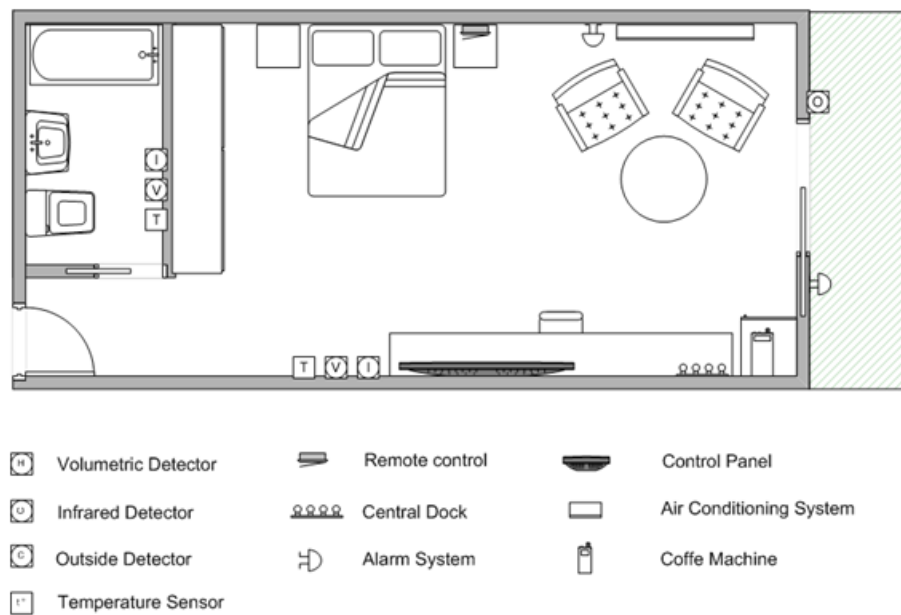


Figure A.1: Hotel's Smart Room

We can observe a possible recreation of the room's hotel in figure B.4 with all the devices that it could include. For example, on the rooms main desk we can see the central dock that the user can interact with to charge any of his own devices or synchronize them with the room control panel in order to check his schedule or personal information. The multi-touch control panel allows the professor to change any of the room's features (illumination, temperature, etc...) and also check any kind of information that could be useful for him (transport, city guide, restaurants and so on). These and the rest of the devices will be described in the following sections.

A.2 Scenarios of the Smart Hotel

This section offers a brief introduction of all the scenarios that will be described in this chapter. All the scenarios try to cover all the possible situations that can occur in the hotel's smart room. The descriptions show a motivation. This refers to all the things that need to be changed in the room to adapt to each scenario.

- **Check-In.** When the user registers (online from the internet or at the hotel's reception desk), he is provided with a wizard that makes a few questions to

set up the room according to the professor's preferences.

Motivation: Reconfigure the room according to the preferences of each user.

- **Entering the room.** When someone enters in the room, it detects all the devices that the user is travelling with. The room services reconfigure to integrate all these devices.

Motivation: Reconfigure the room to integrate the devices in the environment.

- **Activity.** The room reconfigures itself according to the activities that the user performs in it. The activities can be working, watching a movie or sleeping.

Motivation: Reconfigure the room services according to the activity that the user is performing.

- **Leaving the room.** When the professor leaves, the room is reconfigured disabling the services that are no longer needed. Because no one is in the room, it reconfigures itself in order to save energy. The room takes into account when the user has planned to come back (agenda) so he can find the room in his preferred conditions (illumination, temperature...).

Motivation: Save energy while there are no users in the room leaving the room prepared so the user can find everything as he expects when he comes back.

- **House Keeping.** The room reconfigures itself to make the work easier for the cleaning service, at the same time maintains the user's privacy in the room.

Motivation: Make the work for the cleaning service easier and at the same time maintain the room's user's privacy.

- **Check-Out.** Finally, when the user finishes his stay in the room, it stops being personalized for that user and its services are reconfigured in order to save energy.

Motivation. Reconfigure the room to an energy saver mode for the periods when there is no user using it.

By combining the scenarios introduced above it is possible to describe Professor John's stay at the hotel. These scenarios can be combined in different ways. An

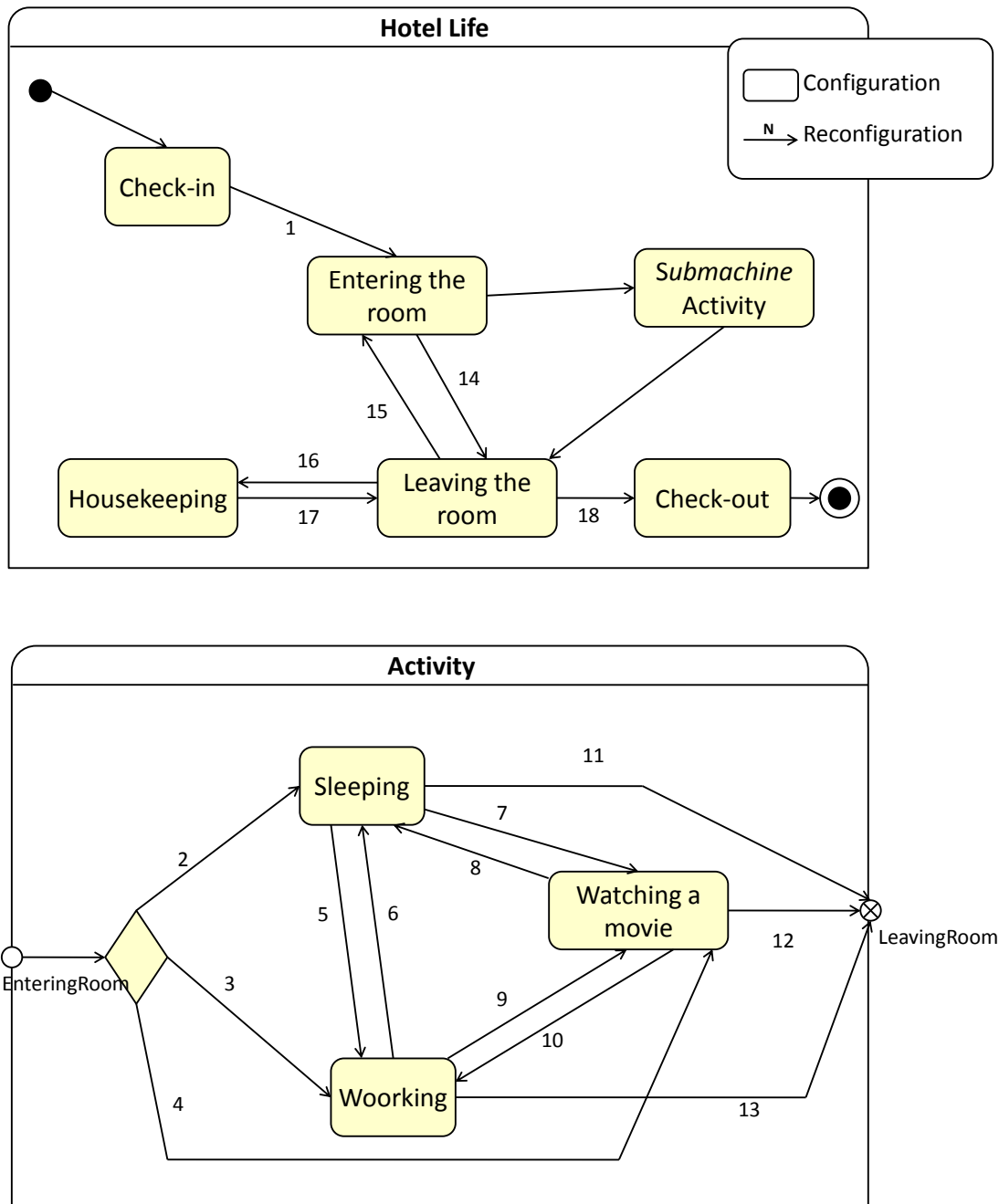


Figure A.2: Reconfiguration through the Smart Hotel Scenarios

example of combination that describes the stay of Professor John could be like the following one:

The user checks-in in this hotel (**Check-in**) through the website, by telephone or even at the reception desk. When he receives his room's card, he can immediately enter his room. When he enters the room(**Entering the room**), he has some free

time and he decides to watch a movie selecting one from the hotel's pay per view service (Activity - Watching a movie). It has become late so, after watching the movie the user decides to go to sleep (Activity - Sleeping). The next morning, the system will wake him up at the time he has scheduled. After preparing everything, the user leaves the room (Leaving the room).

During the user's absence, the hotel's cleaning service proceeds to the room's maintenance (House Keeping). When the user comes back from his conference (Entering the room), he has to pack everything to go back to his country. When everything is prepared, he leaves the room (Leaving the room) and after that he checks-out at the hotel's reception desk (Check-out).

Figure A.2 uses the notation of the state machine to show how the scenarios are related. The scenarios are represented by states and the transitions indicate that it is possible to move from one scenario to another. For example, once the user has left the room (leaving the room), the room can move to the House Keeping, Entering the room or Check-out scenario.

According to the state machine in Figure 2, the stay of the professor described in this section would be as follows.

Check-in → Entering the Room → Watching a Movie → Sleeping → Leaving the room → House Keeping → Leaving the Room → Entering the Room → Leaving the Room → Check-out

In the following sections, we will describe the scenarios denoted in the state machine and how are supported all the different reconfigurations between the scenarios.

A.3 Functionality of the Smart Hotel

Figure A.3 represents the smart hotel's functionality and its possible variations using the notation of the feature models. It shows different squares coloured in gray and white that represents the active and the inactive features in the smart hotel's room, respectively.

The Feature Model intent is to represent all the different features that the Smart Room has implemented to make the user in that room feel as comfortable as possible.

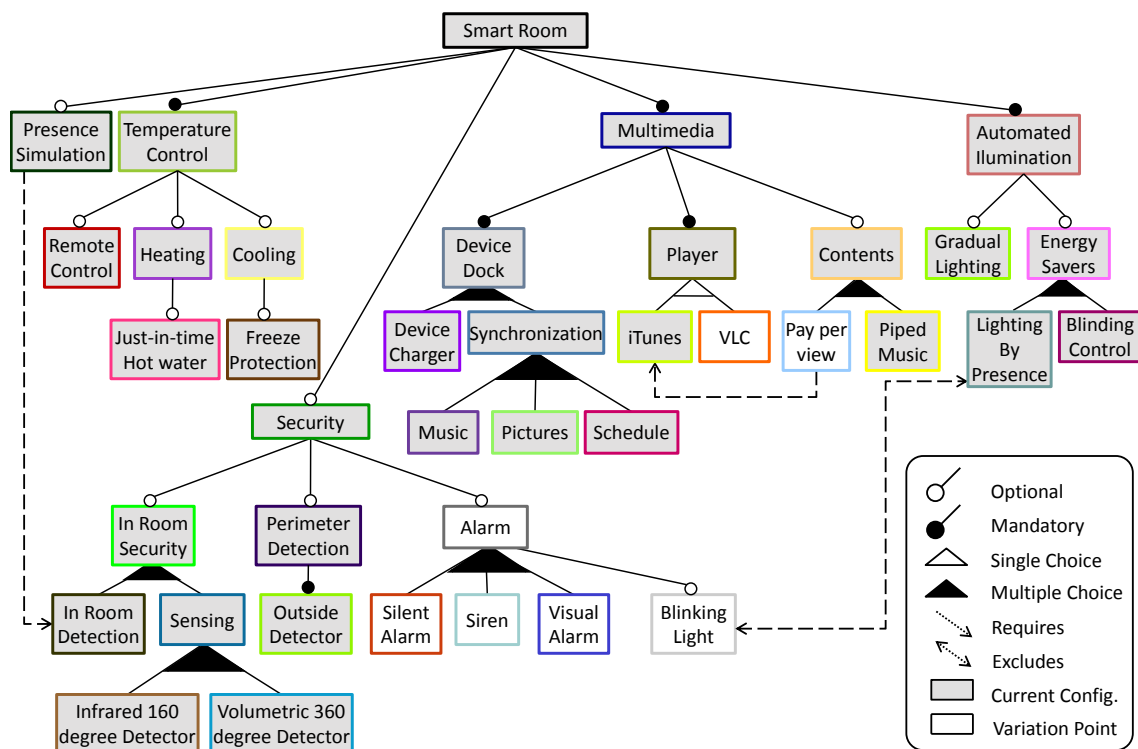


Figure A.3: Feature Model of the Smart Hotel

The feature model changes its features, activating and deactivating them, depending on the scenario that is currently running. That is, the grey features represent the features of the smart home, while the white features represent potential variants since they may be activated in the future.

The Feature Model of Figure A.3 describes a Smart Hotel with Temperature Control, Automated Illumination, Multimedia and Security. These features are hierarchically linked in a tree-like structure through variability relationships such as optional, mandatory, single-choice and multiple-choice as illustrated in Figure A.3.

A.4 Software Architecture of the Smart Hotel

In order to provide a flexible reconfiguration, the smart hotel architecture is based on different components with communication channels. We classify these components into two categories: Services and Devices. This architecture allows an easy reconfiguration since communication channels can be established dynamically between the components, and these components can dynamically appear or disappear

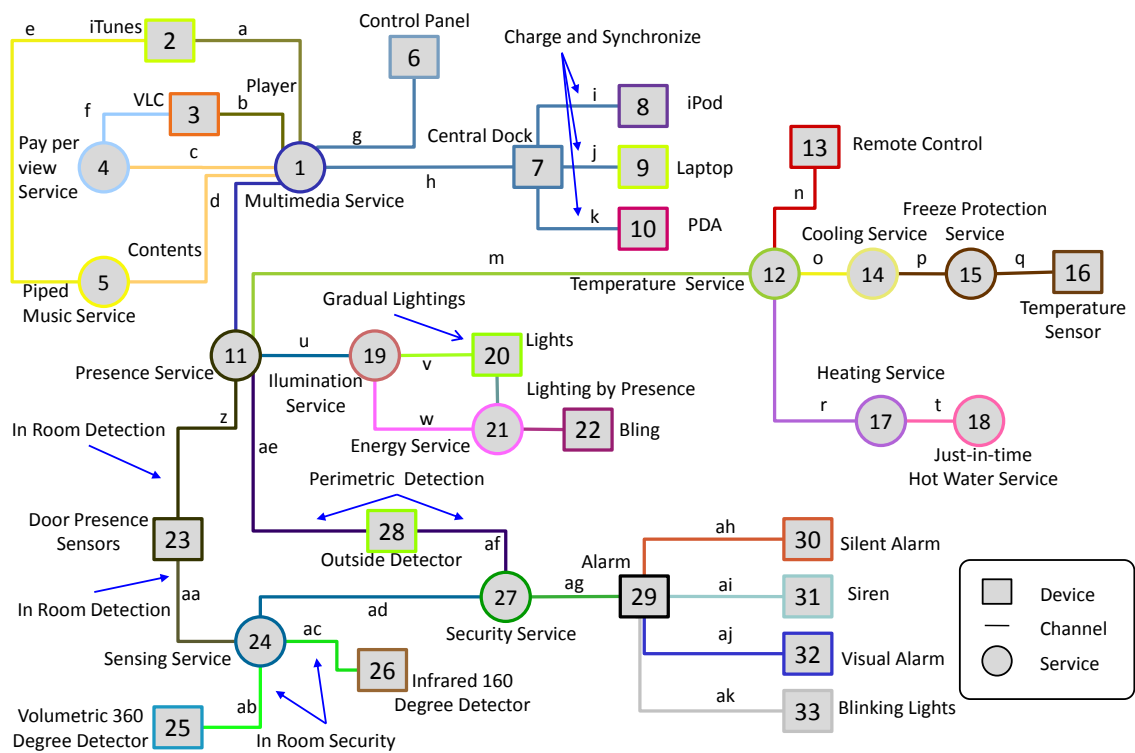


Figure A.4: PervML Model of the Smart Hotel

from configurations. Figure 4 shows this reconfigurable architecture according to the concrete syntax of the PervML Domain-Specific Language. Services are represented by circles, and Devices are represented by squares. Finally, the channels among services and devices are depicted by lines.

In the following examples we can appreciate how some of the features of the feature model are related to the PervML scheme which is known as superimposition according to SPL terminology.

Figure A.5 shows the correspondence between the Feature Model and the PervML scheme. It shows specifically the equivalence from the part of the video and audio players available in the room (VLC and iTunes) connecting them directly with the multimedia service. The Multimedia feature is related with the Multimedia service, the Player feature is related to the Player channels and the iTunes and VLC are related to the iTunes and VLC devices.

Figure A.6 shows another correspondence between the Feature Model and the PervML scheme. The cooling feature is related with the Cooling Service and the Freeze Protection feature is related with the Freeze protection service and the room's

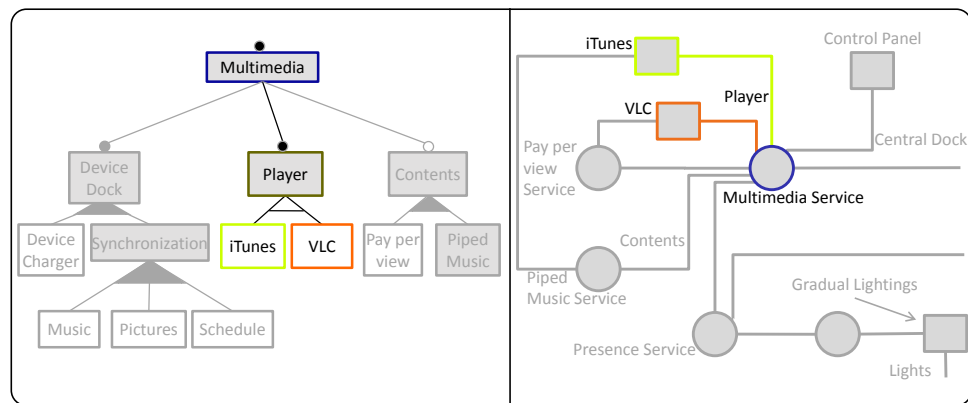


Figure A.5: Mapping between features and PervML

Temperature Sensor.

A.5 Reconfigurations in the Smart Hotel

This section describes a possible stay of Professor John in the Smart Hotel. The stay will begin with the check-in scenario where the professor needs to book for the room and, if he wants, set up all the preferences he wants to have when he enters the room.

Once he arrives at the room, everything will be prepared as he has chosen in the check-in scenario. The room will reconfigure for whatever he wants to do. It will adapt itself automatically when he has to work, watch a movie, sleep or if he has to leave for some conference or any other issue. When he is out, the hotel's cleaning service proceeds to the house keeping maintaining at that moment Professors privacy. When Professor finishes his stay, he leaves the room and proceeds to check-out at the hotel's reception desk

To describe each scenario the following points are included: Description, Feature Model for that configuration and how the services, devices and channels are at that moment.

Apart from that, in order to describe the reconfiguration between the scenarios, some tables describe the following information: Categories, Description, Reconfiguration Trigger, Functionality, Architecture Increments, Architecture Decrements and, with the help of the PervML schemes, the change that will be produced when

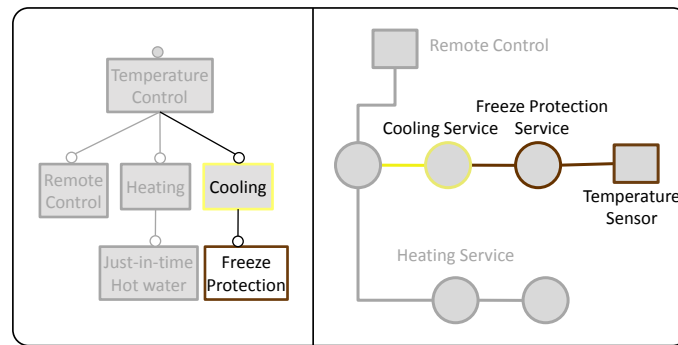


Figure A.6: Another mapping between features and PervML

changing to the current scenario.

A.5.1 Check-in

The following subsection offers detailed information of the Check-in scenario, explaining when that scenario activates, all the user's actions or devices that produces the change and finally a list of all the devices that take part in this scenario.

Description

A few months earlier when the attendance to the congress is confirmed, Professor is going to book the hotel and then set up the room to his own preferences. The hotel allows configuring and customizing a wide amount of options. Not only the room's preferences, activities and services that the hotel offers, but also the activities that can be done in the city.

To obtain this level of satisfaction that will make the Professor to be completely satisfied with his stay, before that, some kind of parameters must be selected: the main point would be configuring the room's environment. Professor can select the kind of room he prefers within the options that the hotel offers him. Professor can select the temperature and the light intensity he prefers. Anyway, he can choose and change all this parameters anytime when he is in his room.

Later, Professor can set-up his working environment, introducing by direct typing or also synchronizing his own devices (PDA, laptop, etc...) with the hotel's booking system. This way, the hotel will organize Professor's working hours. For sure,

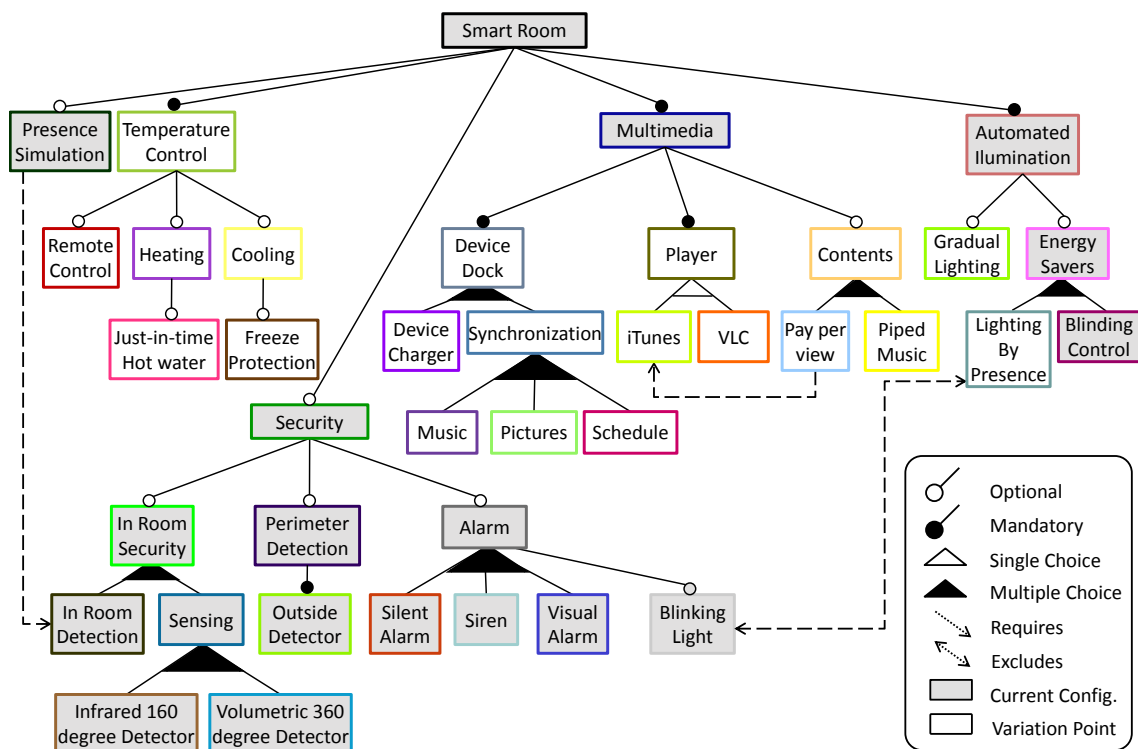


Figure A.7: Feature model of the Check-in Scenario

schedules, tasks, conventions and so on can change until the day of the convention. The same would be for Professor with the things he wants to do during his stay. Because of these issues, the system can be updated through the website. Even when the user preferences change the day before the arrival, Professor can synchronize the moment he enters the room and the room's system will redistribute all the tasks.

Another system option allows sending documents or digital files that the user can require in his job. These files can be modified and the system will keep the different versions until the last day the professor stays in the hotel.

Finally, Professor can set his recreational or free time options: he has a movie library, TV shows library or musical albums library completely customizable through file transferring or, if not, requesting them through a form. The same way, all the free time options (normally sports or cultural) that the hotel organizes are displayed. He will also be suggested with different gastronomic and touristic routes around the city, so he can enjoy his stay as much as possible. Those routes are available to be downloaded and can be integrated with different geolocation systems, street guides, gastronomic guides, etc...

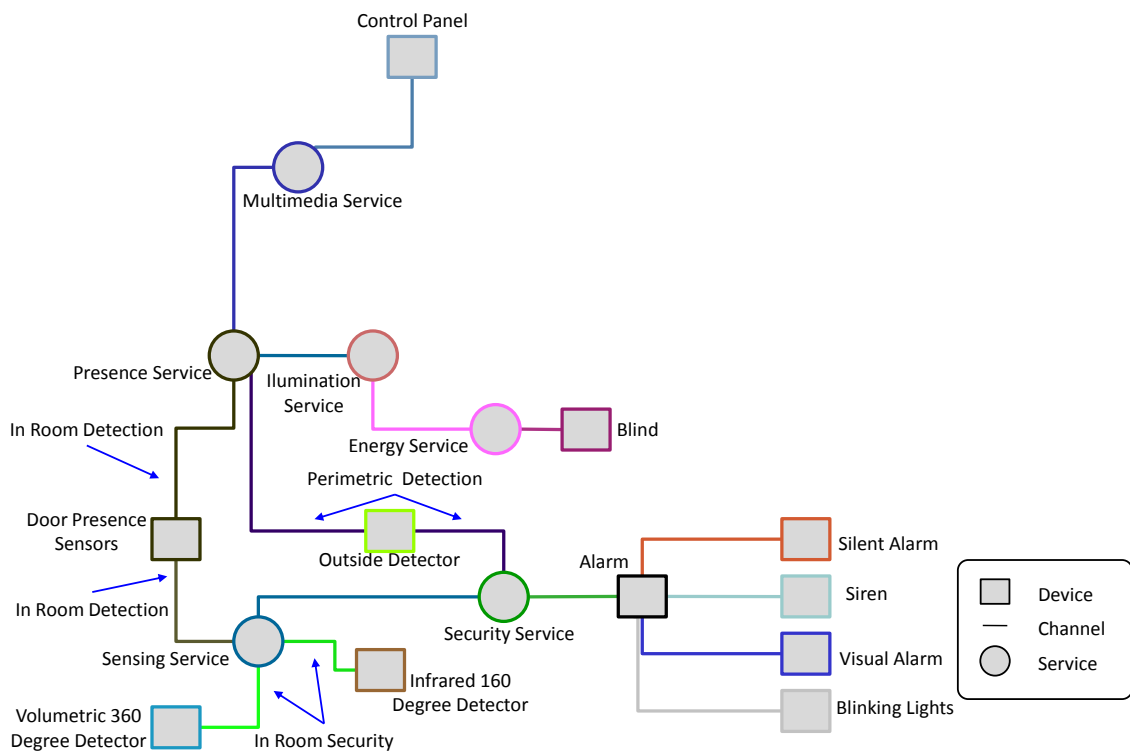


Figure A.8: PervML model of the Check-in Scenario

All the Professor's preferences can change at any time before the arrival at the hotel, or even there manually or automatically, depending on the preferences that professor has introduced in his own electronic working devices.

At the end, the configuration will be saved and also all the changes that have been done for professor's future visits.

Devices involved in the check-in scenario.

- Internet connected terminal to do all the booking and set-up process.
- Agenda system (PDA, cell phone, computer software) connected to the terminal(wired, Bluetooth or Wi-Fi)
- Multimedia system with file library (iTunes, iPod, amarak, windows media player, etc...).
- Control panel to perform basic tasks related with the room such as raising or lowering the blinds or setting the light intensity.
- Blinds which feature a drill motor to automatically roll themselves.

- Presence sensors detect the minute flexing caused by someone walking on the room surface.
- Volumetric detectors are used to detect presence of people in an area. It is designed to be recessed into a ceiling space and can be installed individually in a small room or in groups to cover a larger area.
- Outside sensors features a photocell and they are used to determine light level in an area.
- An audible or visual alarm to alert people of critic notifications such as fire or water leaks in the room.
- Lighting devices ranging from ambient lights (suited for regular activities such as working) to colour-based led lights (suited for entertainment activities such as watching a movie).

Figure A.7 shows the feature model with the active and the inactive features in the check-in scenario, and Figure A.8 shows the PervML model corresponding to the Check-in scenario.

The next table shows the reconfiguration process when the room changes from another scenario to the check-in scenario but since the check-in is the start point of this case study, there is no previous scenario to come from. We consider that all the devices are activated when the check-in scenario is set.

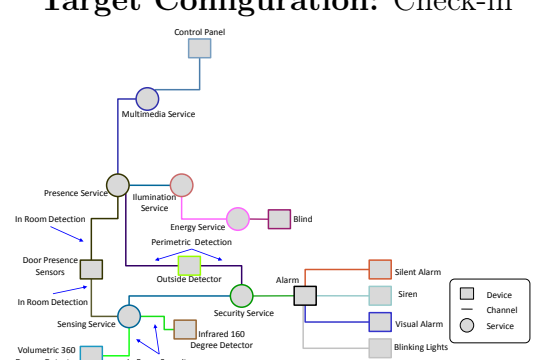
<p>Code: SH-01</p>	<p>Title: Check-in</p>
<p>Categories: Self-configuring, Self-adapting</p>	
<p>Description: The user sets up the entire configuration for the room as he wants (temperature, illumination, etc. . .) In the check-in scenario, there is no previous scenario to come from.</p>	
<p>Reconfiguration Trigger: The user checks-in through webpage or talking to the hotel’s reception desk.</p>	
<p>Reconfiguration Effect: The alarm system, blinds and control panel are enabled.</p>	
<p>Functionality={(Control Panel, True), (Multimedia Service, True), (Presence Service, True), (Illumination Service, True), (Gradual Lighting, True), (Lights, True), (Energy Service, True), (Blind, True), (Perimeter Detection, True), (In Room Detection, True), (Doors Presence Sensors, True), (Outside Detector, True), (Sensing Service, True), (Volumetric 360 Degree Detector, True), (Infrared 160 Degree Detector, True), (In Room Security), (Security Service, True), (Alarm, True), (Silent Alarm, True), (Siren, True), (Visual Alarm, True), (Blinking Lights, True)}</p>	
<p>Architecture Increments: 1, g, 6, l, 11, u, 19, w, 21, s, 22, z, 23, aa, 24, ab, 25, ac, 26, ad, 27, ae, 28, af, ag, 29, ah, 30, ai, 31, aj, 32, ak, 33</p>	
<p>Architecture Decrements: -</p>	
<p>Source Configuration: -</p> <p>No source configuration.</p>	<p>Target Configuration: Check-in</p> 

Table A.1: Reconfiguration Table: Check-in Scenario.

A.5.2 Entering the room

The following subsection offers detailed information of the Entering the Room scenario, explaining when that scenario activates, all the user's actions or devices that produces the change and finally a list of all the devices that take part in this scenario.

Description

When Professor arrives into the room, he must find it like he specified in the configuration he made months before when he booked (or like he changed until the previous day of his arrival). Because of these specifications, all the room parameters like temperature, light intensity, humidity and so on, are as Professor specified. Also, because Professor comes from a South-European country, his room will be located in a place oriented to the south. This way, it will avoid the (first sunlight of the area) (it is earlier than in Professor's home country) but can enjoy the light and the natural warmth during most part of the day.

Once he gets into the room, the system will welcome him. It can be through the systems main screen that can be seen from anywhere in the room or combining those images with a voice (the system's voice will only be available in a few languages).

When professor needs, he has a Wi-Fi system available to connect to the internet anytime he wants. He also has the option to connect all his devices through a central dock and at the same time he is able to recharge them with the same dock.

The system informs the user that his agenda and files he has available in his devices can be synchronized in order to plan the tasks he has to do during his stay. During the process, the system will inform Professor of the stored data to check if everything is up to date or if something needs to be changed because there has been a schedule modification recently.

Depending on Professor's planning, the system will indicate professor about his near appointments or will start the chime and thanks to the thermal and movement detectors, will describe each place of the room that professor goes for the first time. Of course this option can be disabled at anytime.

If professor wishes, he can synchronize his devices through a central dock that

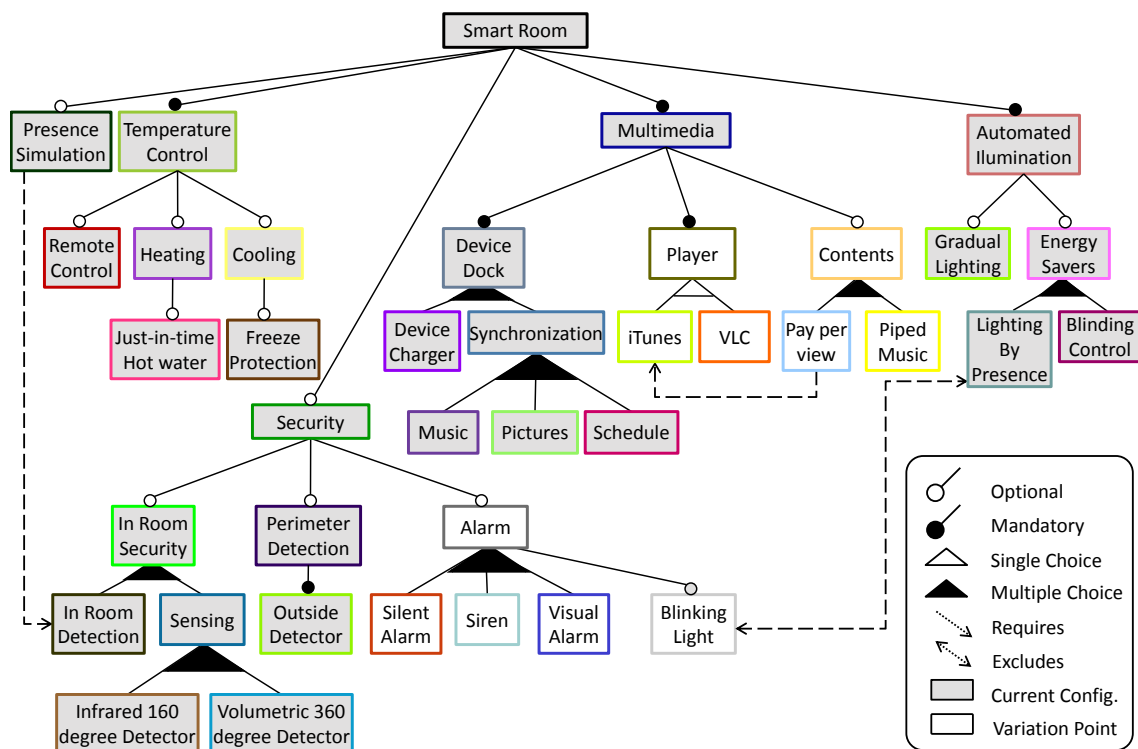


Figure A.9: Feature model of the Entering the room Scenario

will also allow him to recharge its batteries.

The environment is reconfigured with the illumination and the air conditioning system as the time goes like Professor specified so it changes to his liking.

Devices involved in the entering the room scenario.

- Agenda system (PDA, cell phone, computer software) connected to the terminal(wired, Bluetooth or Wi-Fi)
- iPod or multimedia device with streaming capabilities to play music and wired or wify connexion.
- Laptop which stores the personal files of the user and his schedule and appointments.
- Room's control panel (multi-touch TFT screen) which integrates the control of the main services of the room.
- Central dock to synchronize devices and charge batteries by means of wired or wireless (synchronize only) technologies.

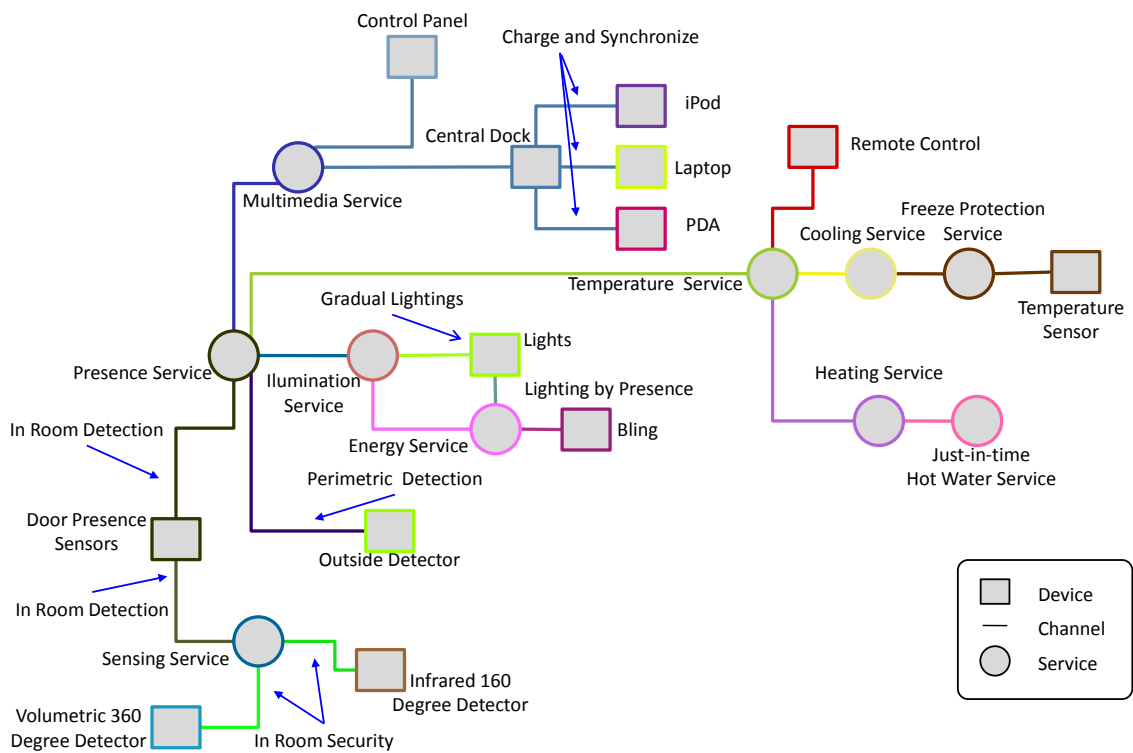


Figure A.10: PervML model of the Entering the Room Scenario

- Temperature sensors are based on infrared technology to keep track of temperature levels in the room.
- Remote controls or thermostats regulate the temperature of a system so that the temperature of the room is maintained near a desired setpoint temperature set by the user.

Figure A.9 shows the feature model with the active and the inactive features for the Entering the Room scenario, and Figure A.10 shows the PervML model corresponding to the Entering the Room scenario.

The next table shows the reconfiguration process when the room changes from the Check-in scenario to the Entering the Room scenario.

Code: SH-02	Title: Entering the Room
Categories: Self-configuring, Self-adapting	
Description: This scenario occurs after the user checks-in or leaves the room.	
Reconfiguration Trigger: The door's identification device validates the user's hotel card and activates all the services while opening the door.	
Reconfiguration Effect: The system enables the air conditioning service, the central dock (for charging and synchronizing), remote control and light functionality. The alarm system is disabled.	
Functionality ={(Central Dock, True), (iPod, True), (Laptop, True), (PDA, True), (Temperature Service, True), (Remote Control, True), (Cooling Service, True), (Freeze Protection Service, True), (Heating Service, True), Just-in-Time Hot Water Service, True), (Lights, True), (Alarm, False), (Silent Alarm, False), (Siren, False), (Visual Alarm, False), (Blinking Lights, False)}	
Architecture Increments: h, 7, i, 8, j, 9, k, 10, m, 12, n, 13, o, 14, p, 15, q, 16, r, 17, t, 18, v, 20, x	
Architecture Decrements: af, ad, 27, ag, 29, ah, 30, ai, 31, aj, 32, ak, 33	
Source: Check-in 	Target: Entering the room

Table A.2: Reconfiguration Table: Entering the Room.

A.5.3 Working

The following subsection offers detailed information of the Working scenario, explaining when that scenario activates, all the user's actions or devices that produces the change and finally a list of all the devices that take part in this scenario.

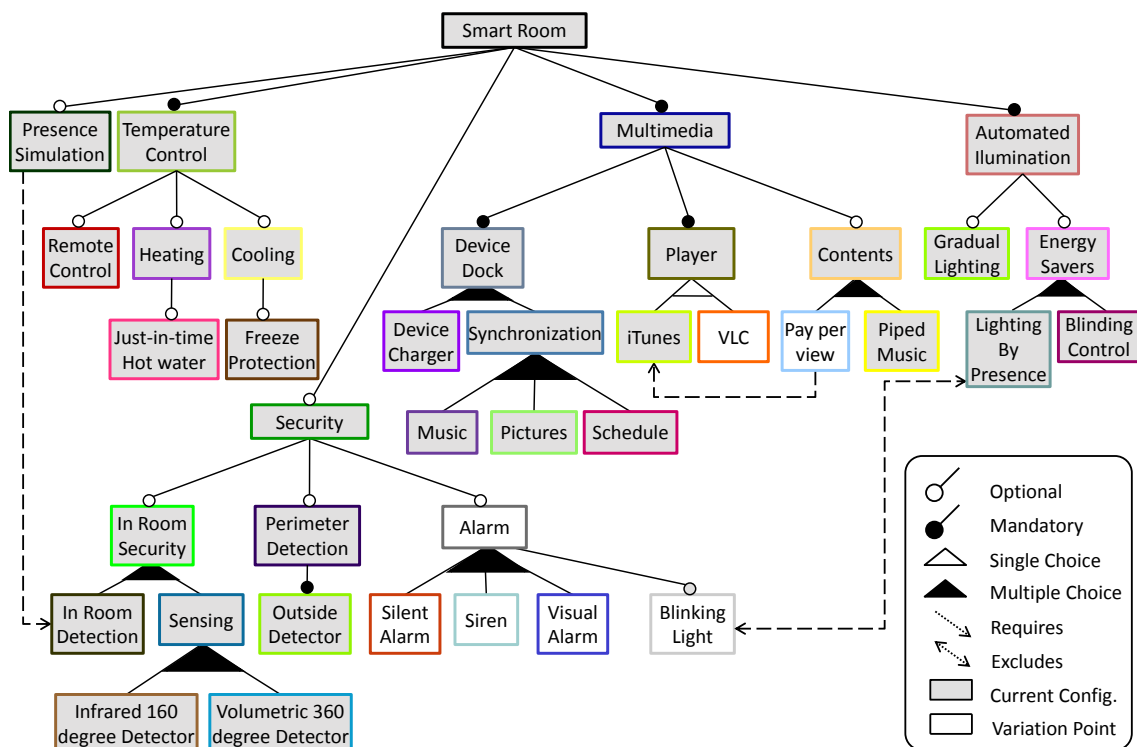


Figure A.11: Feature model of the Working Scenario

Description

It's time for Professor to work on the presentation he has to do for the next day so, when he activates the working mode, the system will ask him for his preferences for having the best working environment. He can modify the room's temperature and illumination and also some suggestions will be offered in case he doesn't know what could be better. If he wants, he also has the option to listen to the music contained in his own devices or listen to the music the hotel has. He will be able to choose the genre of music or the artist or group he prefers to listen while working.

Professor can choose if he wants to be notified when he receives a call or if he has an appointment in his schedule.

Unfortunately Professor couldn't finish all the work on time and he needs more than expected before he goes to the meeting. In this case, the system will suggest him to send a mail automatically to the people that will attend to that meeting to inform that he will be a bit later than expected.

Devices involved in the working scenario.

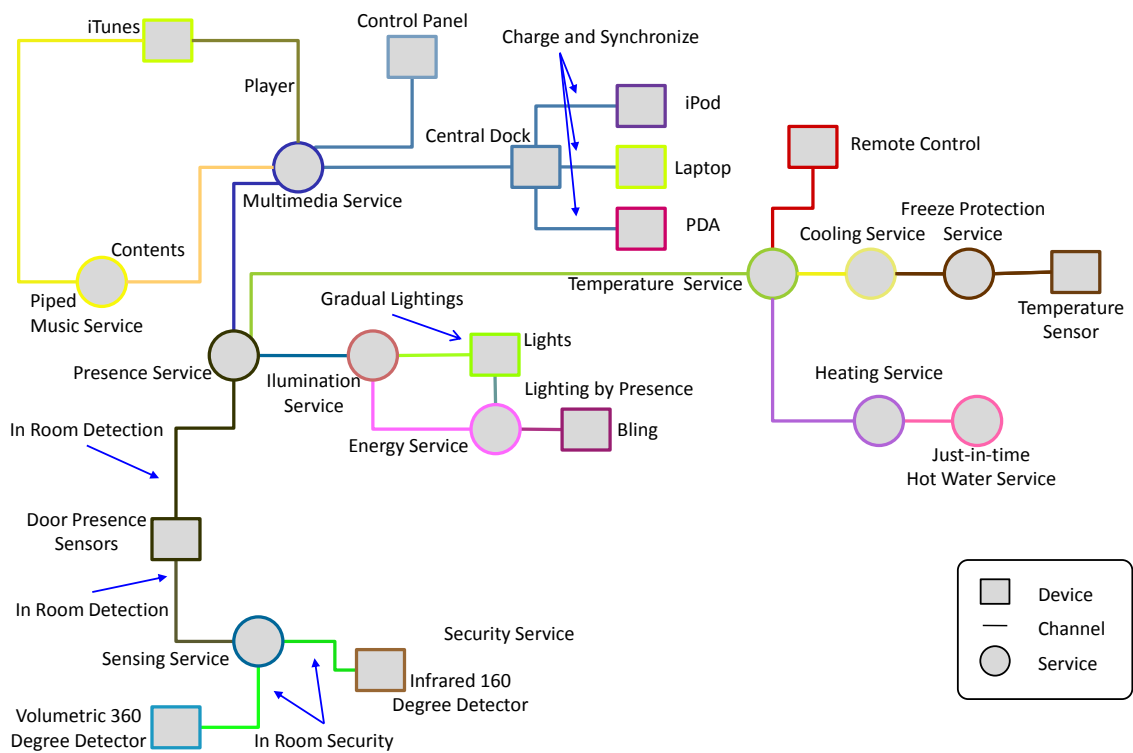


Figure A.12: PervML model of the Working Scenario

- Audio device like iPod to listen music through the room's sound system.
- Cell phone with Bluetooth technology in case the user receives a call and wants to be notified.
- Notebook or cell phone synchronized with the room's main system to notify the user any appointment on his schedule.
- Central dock to synchronize devices and charge batteries.

Figure A.11 shows the feature model with the active and the inactive features in the Working scenario, and Figure A.12 shows the PervML scheme corresponding to the Working scenario. We can see all the enabled services and devices when this scenario is active.

The next table shows the reconfiguration process when the room changes from the Entering the Room scenario to the Working scenario. As shown in state machine, the current scenario can also come from the Sleeping and Watching a Movie scenarios.

Code: SH-03	Title: Working
Categories: Self-configuring, Self-adapting	
Description: The user can work after entering the room or even after sleeping or watching a movie.	
Reconfiguration Trigger: The desk's and the room presence sensors detect the location of the user sitting on the table. The user sets up in the control panel to enter the working scenario.	
Reconfiguration Effect: The audio service is plays a relaxing music according as the user indicated in the preferences for working.	
Functionality ={(iTunes, True), (Piped Music Service, True)}	
Architecture Increments: a, 2, e, 5, d	
Architecture Decrements: -	
<p style="text-align: center;">Source: Entering the room</p>	<p style="text-align: center;">Target: Working</p>

Table A.3: Reconfiguration Table: Working.

A.5.4 Watching a Movie

The following description offers detailed information of the Watching a Movie scenario, explaining when that scenario activates, all the user's actions or devices that produces the change and finally a list of all the devices that take part in this scenario.

Description

Professor was prepared to go to a meeting he had in his schedule but, unfortunately, the weather was so bad that it was postponed for the next day. Now he has some free time because of that unexpected change. Because he made that change in his

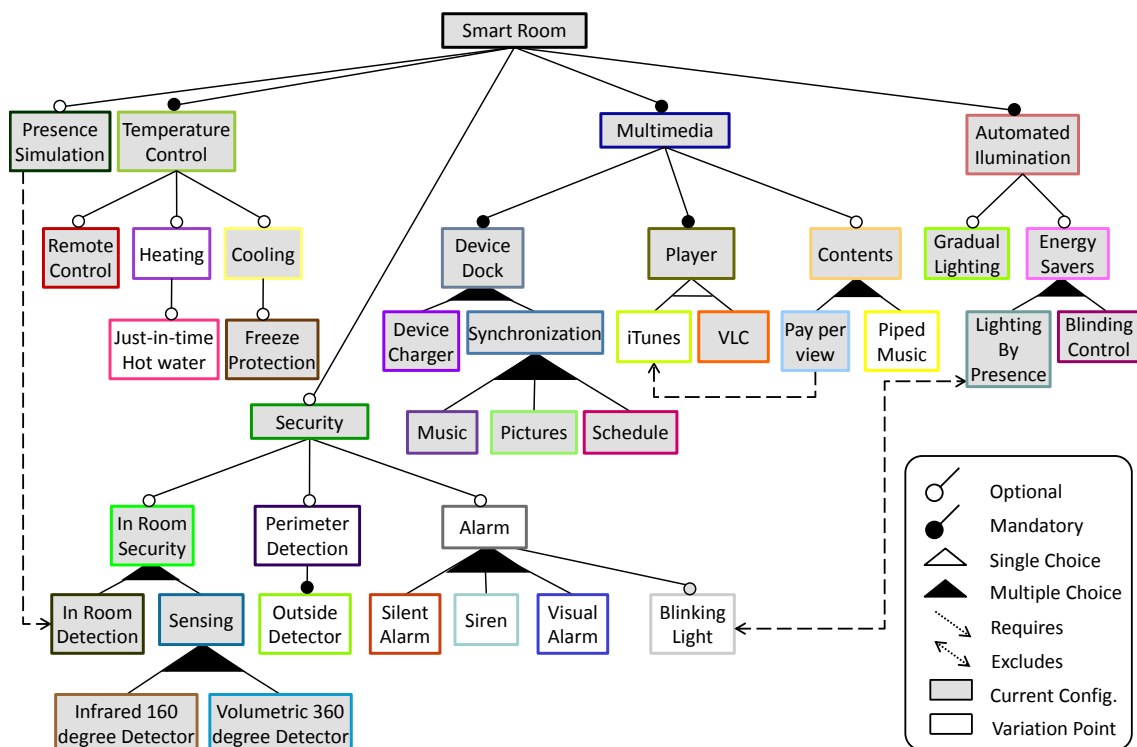


Figure A.13: Feature model of the Working Scenario

schedule, the room system will offer him the option to watch a movie. Professor decides to follow the room's suggestion and is going to watch a movie. The main screen will ask him which kind of movie does he like and, depending on his answer, the system will display all the available movies he can choose. Depending on the kind of movie, he will be able to choose the room preferences (illumination and temperature) for a better experience. The system will give him many options so he can choose the one that suits him more.

Professor will be able to inform the system if he wants to be notified at any time during the movie in case he needs to do something important. He will be able to resume the movie where he stopped at any moment during his stay.

Professor is really enjoying the movie but, unfortunately, someone is calling to his cell phone with Bluetooth technology synchronized with the room's main system. This call will be notified through the main screen. Maybe the call can wait and then Professor has the option to answer it or keep watching the movie where he left it. If he decides to keep watching the movie, he will be asked if he prefers being notified or not in case he receives another call until the movie finishes. Maybe the call is so

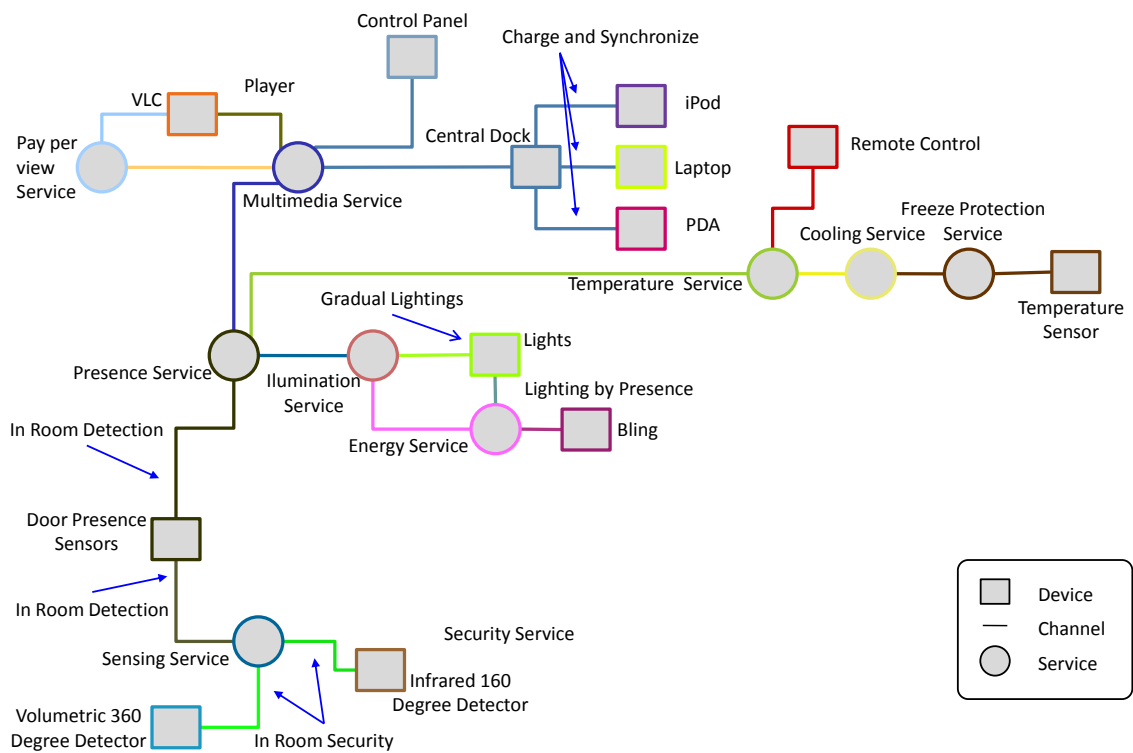


Figure A.14: PervML model of the Working Scenario

important that he has to leave for a while. He is forced to stop watching the movie but he will have the option to continue where he left at any other time during his stay. When Professor is in his room and the system checks in the schedule that he has enough free time, it will remind him that he can try to finish the movie he started. Anyway, Professor will have the option to activate or not to be notified when he receives a call before the movie starts.

Devices in the watching a movie scenario:

- Cell phone with Bluetooth technology in case the user receives a call and wants to be notified. Notify the user any appointment on his schedule.
- Central dock to synchronize devices and charge batteries.

Figure A.13 shows the feature model with the active and the inactive features in the Watching a Movie scenario, and Figure A.14 shows the PervML scheme corresponding to the Watching a Movie scenario. We can see all the enabled services and devices when this scenario is active.

The next table shows the reconfiguration process when the room changes from the

Working scenario to the Watching a Movie scenario. As shown in state machine, the current scenario can also come from the Sleeping and Entering the Room scenario.

Code: SH-04	Title: Watching a Movie
Categories: Self-configuring, Self-adapting	
Description: The user can watch a movie just after entering the room or even after working or sleeping.	
Reconfiguration Trigger: The room’s presence sensors detect the location of the user on the room’s sofa or on the bed. The user selects a movie through the room’s control panel or with the remote control.	
Reconfiguration Effect: The video service is enabled. The audio service and the outside detector are disabled.	
Functionality ={(iTunes, False), (Piped Music Service, False), (VLC, True), (Pay per View Service, True), (Outside Detector, False), (Heating Service, False), (Just-in-Time Hot water Service, False)}	
Architecture Increments: d, 3, f, 4, c	
Architecture Decrements: a, 2, e, 5, d, r, 17, t, 18, ae, 28	
Source: Working 	Target: Watching a movie

Table A.4: Reconfiguration Table: Watching a Movie.

A.5.5 Sleeping

The following subsection offers detailed information of the Sleeping scenario, explaining when that scenario activates, all the user’s actions or devices that produces the change and finally a list of all the devices that take part in this scenario.

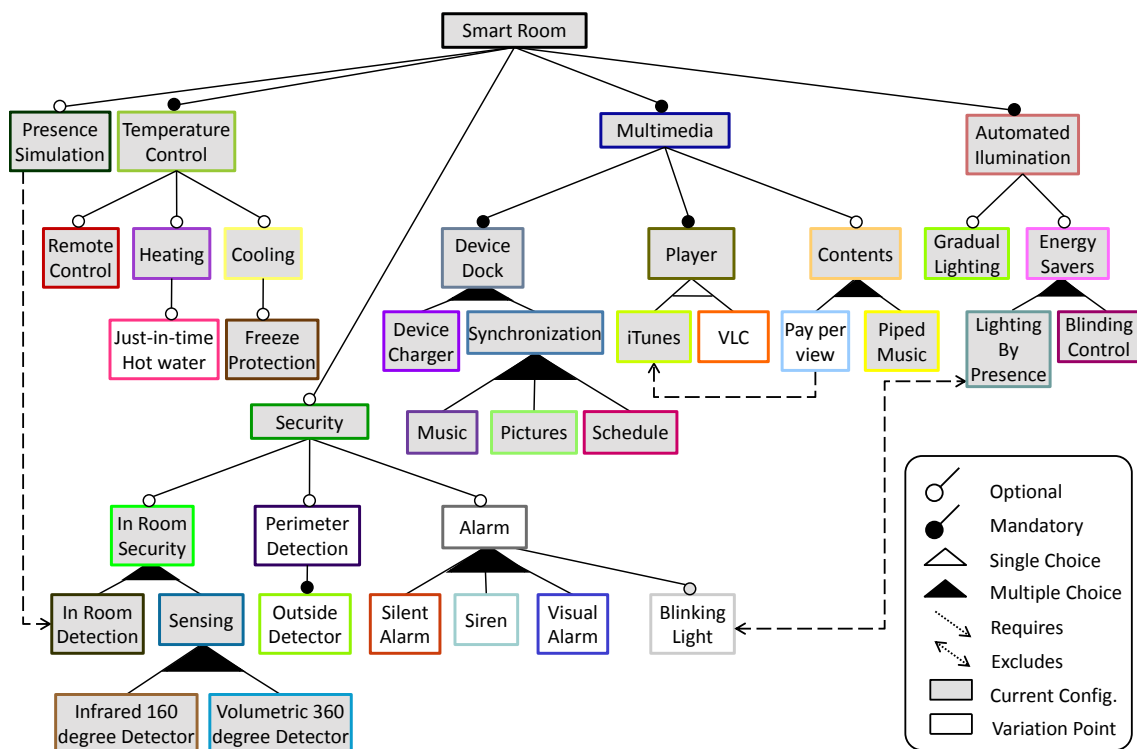


Figure A.15: Feature model of the Working Scenario

Description

The system checks professor's schedule and calculates the time when he should go to sleep so he can be fine the next day. It will suggest him the appropriate time to go to sleep in order to sleep the necessary amount of hours. He decides to follow the advice and then when he sets the system in sleeping mode, the system will decrease the light intensity and the temperature to make the rest more pleasant. If Professor decides to wake up during the night and the room's configuration is in sleeping mode, the lights in the room will turn on lightly so professor can see the room without waking him up completely.

The chime can be played with a timer option with relaxing music and a low volume to increase the sleeping sensation. Relaxing music improves mood compared to traditional loud alarm clocks that tend to jolt the body to wake.

In the same way, all the lights in the in the room will decrease its intensity (unless the sleeping mode is disabled through the graphic interface) to avoid dazzling and waking up Professor.

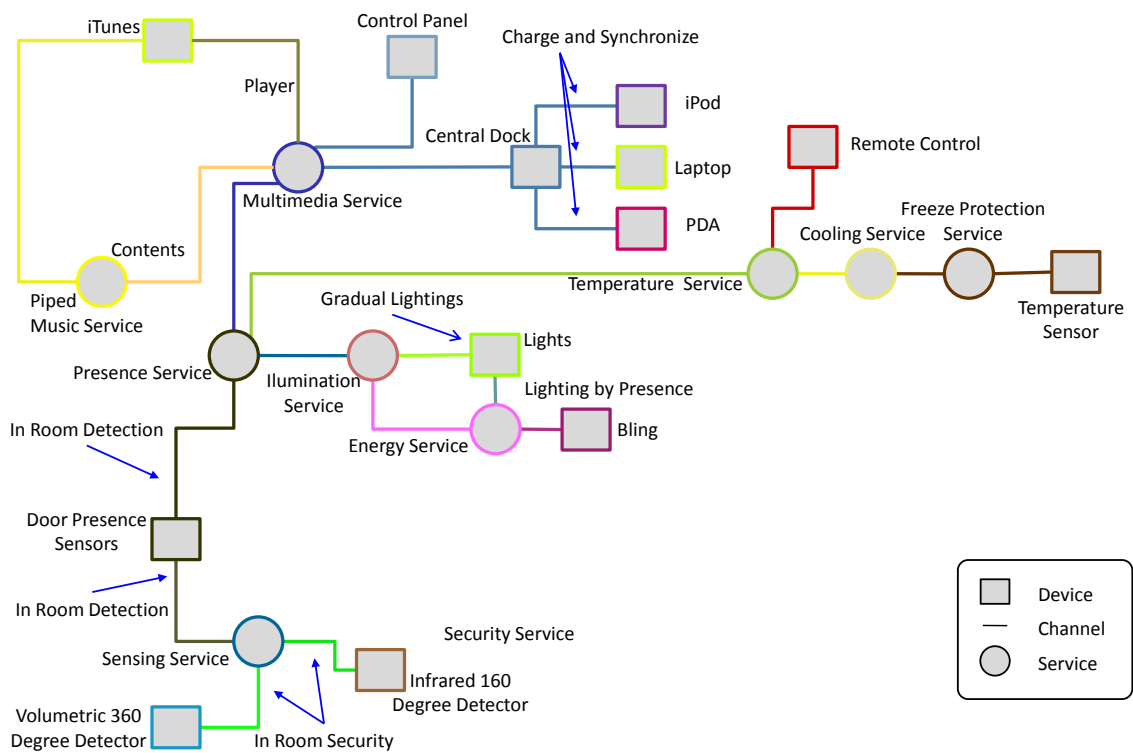


Figure A.16: PervML model of the Working Scenario

Devices involved in the sleeping scenario.

- Room's control panel (multi-touch TFT screen)
- Central dock to synchronize devices and charge batteries.
- Multimedia Manager organizes and plays digital music and video on a computer. In addition, it syncs all the media with external devices.

Figure A.15 shows the feature model with the active and the inactive features in the Sleeping scenario, and Figure A.16 shows the PervML scheme corresponding to the Sleeping scenario. We can see all the enabled services and devices when this scenario is active.

The next table shows the reconfiguration process when the room changes from the Watching a Movie scenario to the Sleeping scenario. As shown in state machine, the current scenario can also come from the Entering the Room and Working scenario.

Code: SH-05	Title: Sleeping
Categories: Self-configuring, Self-adapting	
Description: The user can sleep just after entering the room or even after working or watching a movie.	
Reconfiguration Trigger: The bed sensors detect the user's presence on it and activates sleep mode. The user selects the sleep mode through the control panel or the remote control.	
Reconfiguration Effect: The audio service is enabled. The video service and the synchronizing feature are disabled.	
Functionality ={(iTunes, True), (Piped Music Service, True), (VLC, False), (Pay per View Service, False)}	
Architecture Increments: a, 2, e, 5, d	
Architecture Decrements: b, 3, f, 4, c	
<p style="text-align: center;">Source: Watching a movie</p>	<p style="text-align: center;">Target: Sleeping</p>

Table A.5: Reconfiguration Table: Sleeping.

A.5.6 Leaving the room

The following subsection offers detailed information of the Leaving the Room scenario, explaining when that scenario activates, all the user's actions or devices that produces the change and finally a list of all the devices that take part in this scenario.

Description

Professor needs to leave the room as scheduled. When he leaves the room, it will be reconfigured deactivating all those services that are not necessary in order to

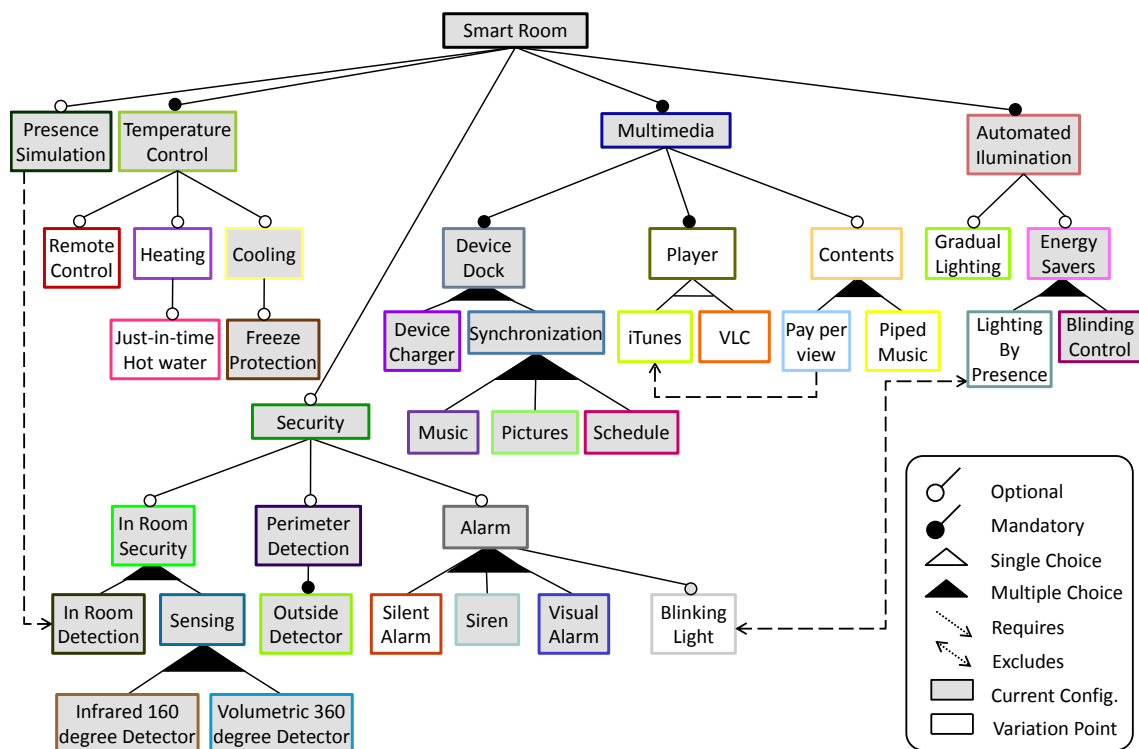


Figure A.17: Feature model of the Leaving the Room

save energy. The room's system keeps in mind when will the user plan to come back checking the schedule he provided when he checked-in. This way, when he comes back, the room will be in the same conditions like when he left (temperature, illumination, etc...). He can also provide information to the control panel to tell if he needs the cleaning service to do something specific while he is out. He can also request other things like presents, movies, and that information can be processed while he is out.

Devices involved in the leaving the room scenario.

- Room's control panel (multi-touch TFT screen)
- Volumetric detectors are used to detect presence of people in an area. It is designed to be recessed into a ceiling space and can be installed individually in a small room or in groups to cover a larger area.
- Outside sensors features a photocell and they are used to determine light level in an area.

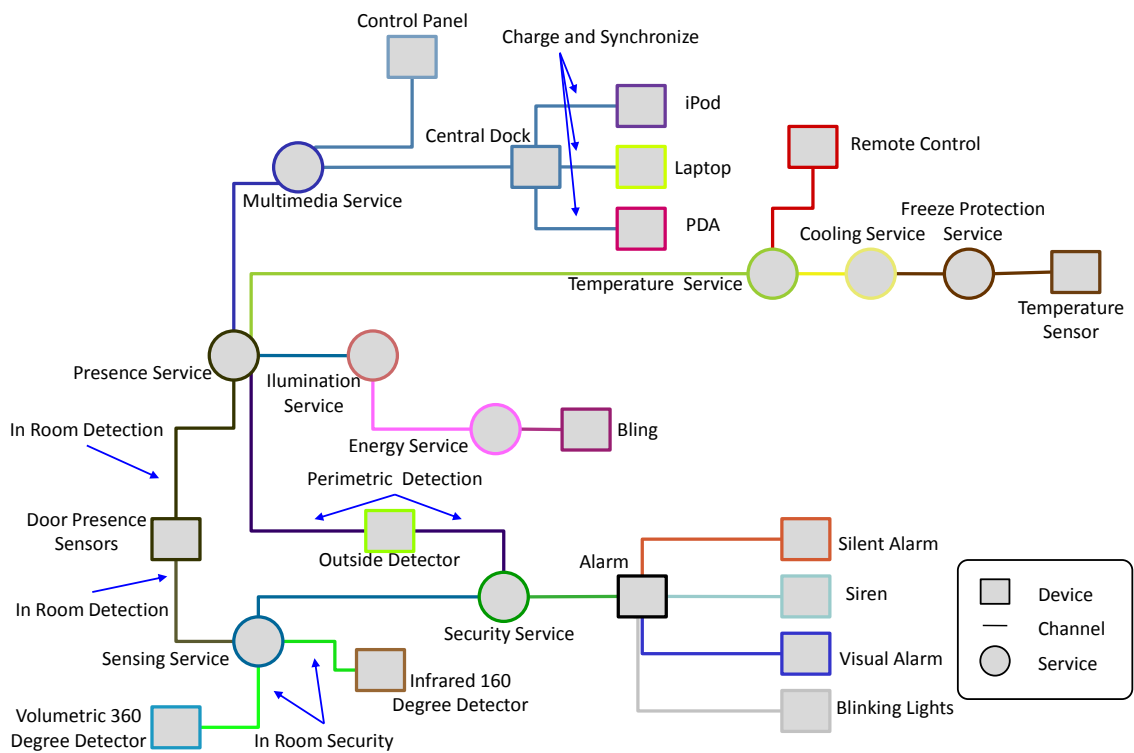


Figure A.18: PervML model of the Leaving the room

- An audible or visual alarm to alert people of critic notifications such as fire or water leaks in the room.

Figure A.17 shows the feature model with the active and the inactive features in the Leaving the Room scenario, and Figure A.18 shows the PervML model corresponding to the Leaving the Room scenario. We can see all the enabled services and devices when this scenario is active.

The next table shows the reconfiguration process when the room changes from the Sleeping scenario to the Leaving the Room scenario. As shown in state machine, the current scenario can also come from the Entering the Room, Working and Watching a Movie scenario.

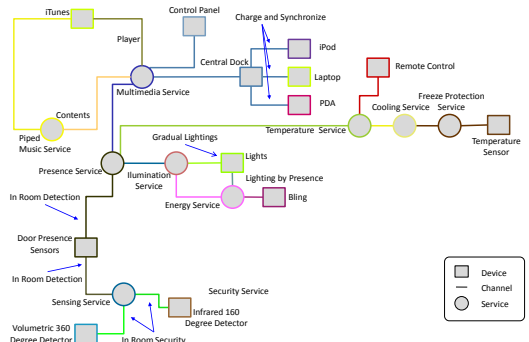
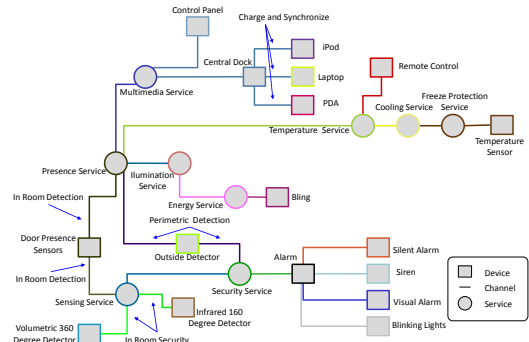
Code: SH-06	Title: Leaving the Room
Categories: Self-configuring, Self-adapting	
Description: In this scenario, the user can leave the room after sleeping, working, watching a movie or even after entering the room. The cleaning service also leaves the room after the maintenance.	
Reconfiguration Trigger: The user opens the door and the room presence sensors stop sensing movement in the room.	
Reconfiguration Effect: The alarm system is enabled. The Audio service, control panel, remote control and lights are disabled.	
Functionality ={(iTunes, False), (Piped Music Service, False), (Control Panel, False), (Remote Control, False), (Lights, False), (Outside Detector, True), (Security Service, True), (Alarm, True), (Silent Alarm, True), (Siren, True), (Visual Alarm, True), (Blinking Lights, True)}	
Architecture Increments: ad, 27, af, ag, 29, ah, 30, ai, 31, aj, 32, ak, 33	
Architecture Decrements: a, 2, e, 5, d, g, 6, h, 13, v, 20, x	
<p style="text-align: center;">Source: Sleeping</p> 	<p style="text-align: center;">Target: Leaving the room</p> 

Table A.6: Reconfiguration Table: Leaving the Room.

A.5.7 House Keeping

The following subsection offers detailed information of the House Keeping scenario, explaining when that scenario activates, all the user’s actions or devices that produces the change and finally a list of all the devices that take part in this scenario.

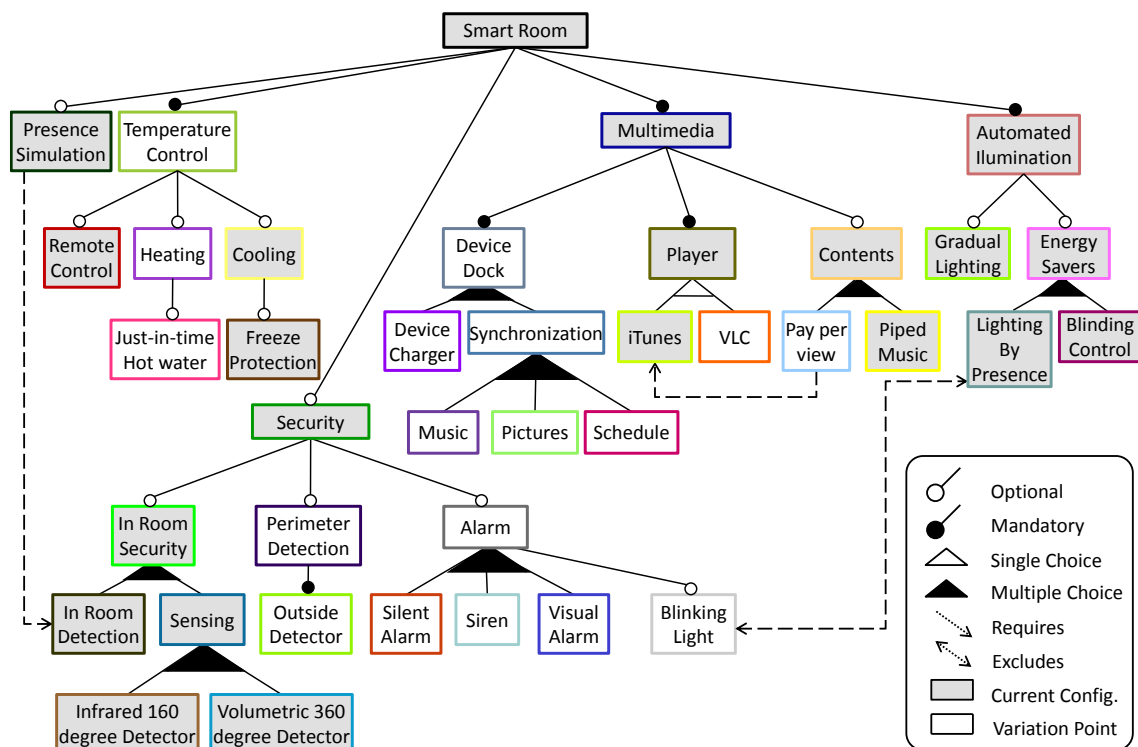


Figure A.19: Feature model of the Housekeeping Scenario

Description

When the user is not in the room, the system will change its set-up to save energy while no one is in there. When the room's cleaning service gets in the room, all the screens will be shut down in order to keep the user's privacy. It won't be possible to access Professors information in anyway while the cleaning service is in the room.

The moment the service gets into the room, the blinds will automatically go up completely because it will be easier for the service to clean everything.

The room cleaning service will have his own PDA to check the time when the Professor will be out and when will he come back. This way they will know how much time they have to clean the room. The room service's PDA will synchronize with the room's system in order to show in the main screen the things they have to do in that room. The system will also notify if the professor has to arrive shortly.

The service also has the possibility to synchronize his own audio device with the room system to listen to his own music. It will help the motivation of the employees while working

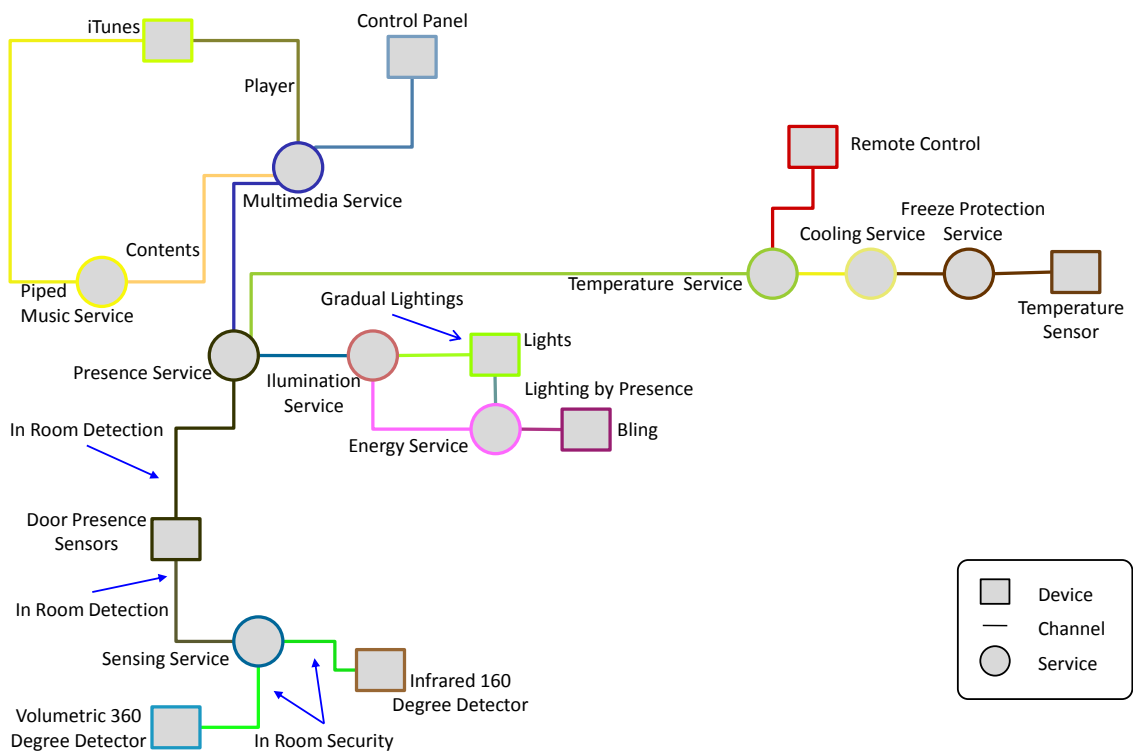


Figure A.20: PervML model of the Housekeeping Scenario

Devices involved in the house keeping scenario.

- Room's control panel (multi-touch TFT screen)
- PDA system connected to the terminal (wired, Bluetooth or Wi-Fi)
- Audio device(iPod, Zune, etc)
- Central dock to synchronize devices and charge batteries.

Figure A.19 shows the feature model with the active and the inactive features in the House Keeping scenario, and Figure A.20 shows the PervML scheme corresponding to the House Keeping scenario. We can see all the enabled services and devices when this scenario is active.

The next table shows the reconfiguration process when the room changes from the Leaving the Room scenario to the House Keeping scenario.

Code: SH-07	Title: House Keeping
Categories: Self-configuring, Self-adapting	
Description: After the user leaves the room, the hotel’s room service can proceed to the maintenance of the room.	
Reconfiguration Trigger: The hotel’s cleaning service enters into the room.	
Reconfiguration Effect: The audio service, control panel, remote control and lights are enabled. The alarm system is disabled.	
Functionality ={(iTunes, True), (Piped Music Service, True), (Control Panel, True), (Central Dock, False), (iPod, False), (Laptop, False), (PDA, False), (Remote Control, True), (Lights, True), (Outside Detector, False), (Security Service, False), (Alarm, False), (Silent Alarm, False), (Siren, False), (Visual Alarm, False), (Blinking Lights, False)}	
Architecture Increments: a, 2, e, 5, d, g, 6, n, 13, v, 20, x	
Architecture Decrements: h, 7, i, 8, j, 9, k, 10, ad, 27, ae, 28, af, ag, 29, ah, 30, ai, 31, aj, 32, ak, 33	
Source: Leaving the room	Target: House Keeping

Table A.7: Reconfiguration Table: Housekeeping.

A.5.8 Check-out

The following subsection offers detailed information of the Check-out scenario, explaining when that scenario activates, all the user’s actions or devices that produces the change and finally a list of all the devices that take part in this scenario.

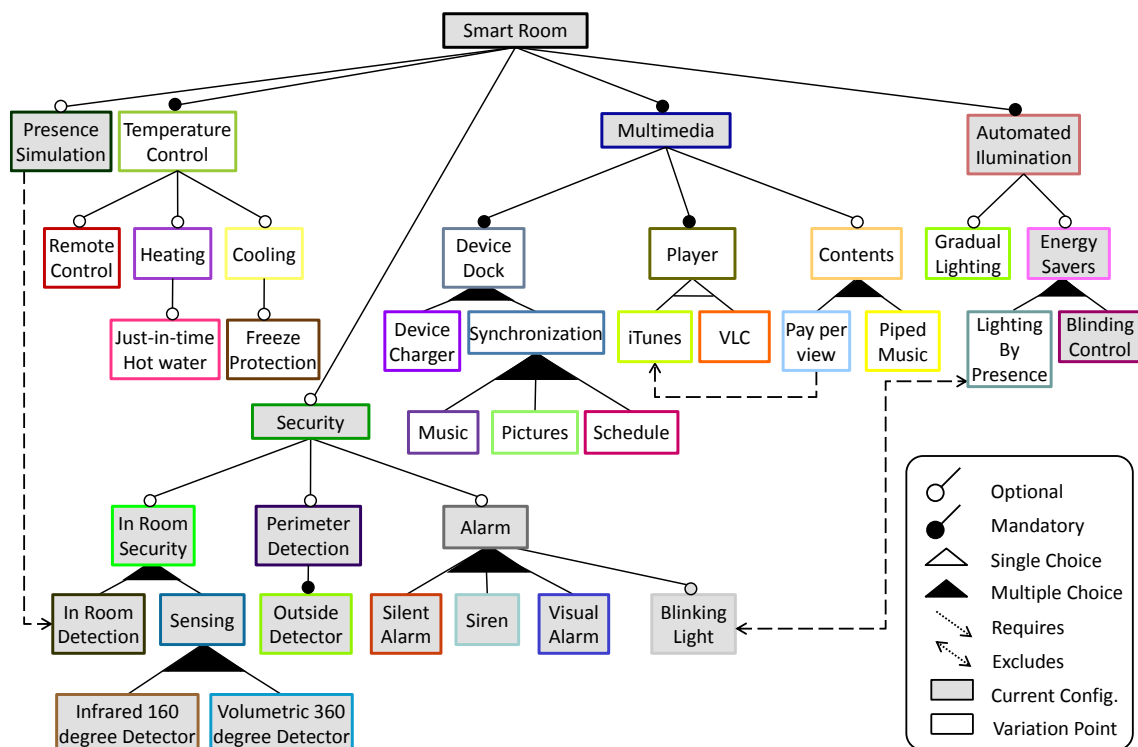


Figure A.21: Feature model of the Check-out Scenario

Description

Finally, it's almost time for the Professor to check-out so the system will notify him during the end of the previous day, at what time does he have to check out on the next day and also what time would be good to leave to be able to arrive to the airport on time. A list of different options of transport to get to the airport or the train station and its own timetables will be displayed so he can choose the option that suits him more.

The day of the check-out, before the Professor leaves the room, the system will ask him if everything was as he liked. He will also have the option to keep his preferences in the hotel servers in case he decides to come back. Anyway, he will be able to change those preferences anytime. The system will also notify the user that all the synchronized information from his devices to the room main system (schedules, musical library, places he went, etc...) will be deleted to keep his confidentiality.

Once the Professor is out, all the electronic systems in the room (screens, lights, air conditioning) will be disabled in order to save energy. During the daylight the

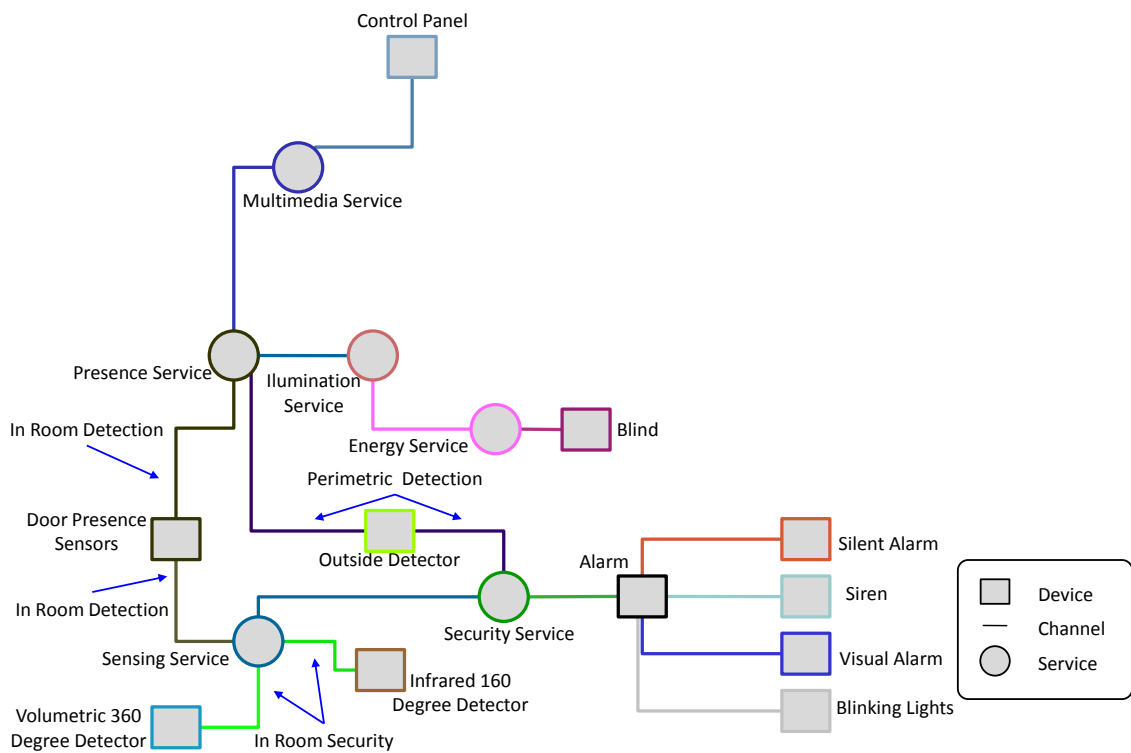


Figure A.22: PervML model of the Check-out Scenario

blinds will rise automatically and will help to keep the room warmer with less energy.

At the reception desk, the user will have to give the room key card back and the hotel's staff wishing him to have a nice trip and hoping he comes back another time in the future.

Devices involved in the check-out scenario.

- The moment Professor checks-out, none of his devices need to interact with the room's system.

Figure A.21 shows the feature model with the active and the inactive features in the Check-out scenario, and Figure A.22 shows the PervML scheme corresponding to the Check-out scenario. We can see all the enabled services and devices when this scenario is active.

The next table shows the reconfiguration process when the room changes from the Leaving the Room scenario to the Check-out.

Code: SH-08	Title: Check-out
Categories: Self-configuring, Self-adapting	
Description: The user finishes his stay in the hotel. He leaves the room and goes to the hotel's reception desk to proceed with the check-out.	
Reconfiguration Trigger: The user performs the check-out at the hotel's reception desk.	
Reconfiguration Effect: The control panel is enabled. The central dock and the air conditioning system are disabled.	
Functionality ={(Control Panel, True), (Central Dock, False), (iTunes, False), (Laptop, False), (PDA, False), (Temperature Service, False), (Cooling Service, False), (Freeze Protection Service), (Temperature Sensor, False)}	
Architecture Increments: g, 6	
Architecture Decrements: h, 7, I, 8, j, 9, k, 10, m, 12, o, 14, p, 15, q, 16	
Source: Leaving the room	Target: Check-out

Table A.8: Reconfiguration Table: Check-out.

A.6 Summary

In this appendix, a case Studio of a Smart Hotel has been presented where the possible scenarios that can occur in the rooms of the mentioned Hotel are specified.

In order to know the functionality and the System's architecture, a Feature Model has been defined. This Feature Model allows to specify the system and all its possible variations. In addition to the Feature Model, the architecture has

been designed using a Domain Specific Language (DSL) called PervML. This model represents all the services, devices and channels that are included in the room to show how they are connected between them.

Each scenario presented in this appendix, has its own Feature Model and PervML model that derive from the general one. This way, all the features, channels, devices and services that are active for each scenario can be specified.

The following offers a brief description of each scenario and also the metrics corresponding to the Feature Model and the PervML model.

- **Check-in.** The user completes the registration of the room, either through the website or at the hotel's reception desk. At this moment, the user can specify the configuration preferences of the room. This scenario is formed with 19 active features in the Feature Model and 17 channels, 11 devices and 6 services in the PervML model.
- **Entering the Room.** Once the user has registered, he enters in the room finding everything like he specified in the check-in scenario. This scenario is formed with 28 active features in the Feature Model and 23 channels, 13 devices and 10 services in the PervML model.
- **Working.** The room reconfigures so the working environment is appropriate for the user. This scenario is formed with 32 active features in the Feature Model and 26 channels, 14 devices and 11 services in the PervML model.
- **Watching a Movie.** When the user decides to watch a movie, the room changes its configuration to improve the viewing experience, adapting the suitable illumination for that purpose. This scenario is formed with 28 active features in the Feature Model and 23 channels, 13 devices and 9 services in the PervML model.
- **Sleeping.** The different sensors in the room detect that the user is going to sleep so the room reconfigures itself to make the user can get to sleep more easily. This scenario is formed with 29 active features in the Feature Model and 23 channels, 13 devices and 9 services in the PervML model.

- Leaving the Room. Once the user leaves the room the system reconfigures in order to save energy. The room will keep the user's desired preferences when he comes back. This scenario is formed with 27 active features in the Feature Model and 24 channels, 15 devices and 9 services in the PervML model.
- House Keeping. The hotel's room cleaning service proceeds to the room's maintenance. The room keeps the user's privacy while he is not in the room. The room's cleaning service can use the room's audio system to listen music while working improving the working environment. This scenario is formed with 22 active features in the Feature Model and 18 channels, 9 devices and 9 services in the PervML model.
- Check-out. Once the user finishes his stay in the hotel, he proceeds to the check-out at the hotel's reception desk. He can keep the room's configuration preferences for the next time he decides to come back. This scenario is formed with 19 active features in the Feature Model and 17 channels, 11 devices and 6 services in the PervML model.

The above scenarios conform the Smart Hotel case study, which is representative of real problems. In addition, this case study has been specifically developed to exercise the reconfigurations of the approach proposed in this thesis, and it has proven itself to be well-understood by users in experimentation.

Since the design of case studies is recognized as a difficult step during the development of experimentation [189], we believe that the Smart Hotel case study can be applied to more empirical research in the context of run-time reconfiguration.

Appendix B. TOOL SUPPORT

The use of variability models for enabling autonomic behaviour is the central idea of this work. At design time, the models that specify the system variability and the system context are built. At run-time, these models are queried in response to context events to produce the system reconfiguration that should be executed. This appendix presents the tool support for both design time (variability and context model specification) and run-time (reconfiguration execution).

B.1 Support for Designing Autonomic Behaviour

Figure B.1 shows the main concepts used at design time to specify the autonomic behaviour and how these concepts are related among them. To enable autonomic system engineers the specification of the autonomic behaviour in terms of these concepts, we provide the following tools.

- **Protege**¹. This tool enables the specification of the system operational environment by means of the *Class*, *Property* and *Instance* concepts. These concepts are described in an OWL ontology as a collection of RDF triples, in which each statement is in the form of (subject, predicate, object). Protege provides a tree editor (see top left of Figure B.2) to specify the former concepts in the OWL ontology.

Protege also provides an expressions editor for describing specific situations in the operational environment of the system (see top right of Figure B.2). This editor supports SPAQRL for the definition of Context Conditions. These

¹<http://protege.stanford.edu/>

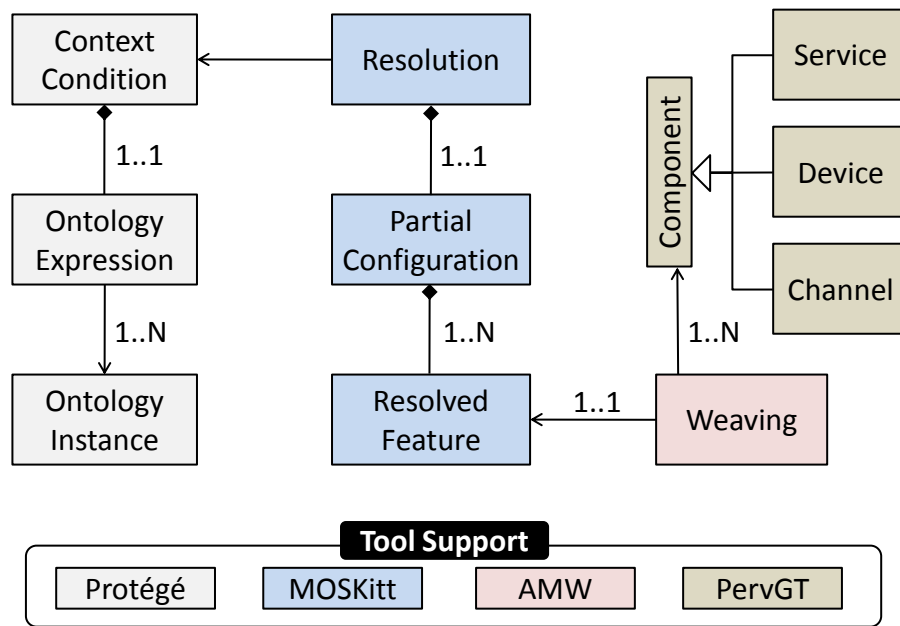


Figure B.1: Tool Support for Design Time.

Context Conditions use an ASK expression to test whether or not a query pattern has a positive solution in the instances of the OWL ontology.

- **MOSKitt**². This tool is a free Modelling platform, built on Eclipse which is being developed by the Valencian Regional Ministry of Infrastructure and Transport. Moskit Feature Modeler (MFM) is the open source feature model editor of Moskitt. MFM enables the specification of the system variability in terms of features, cardinality-based relationships such as optional or mandatory, and cross-tree constraints such as requires or excludes (see center top of Figure B.2).
- **PervGT**³. This tool supports the creation of PervML models. PervML is Domain specific Language for Smart Homes mainly based on the concepts of *Service* and *Device*. PervGT enables the specification of both *Service Models* and *Device Models*. In addition, PervGT support the definition of the Structural Models to establish the relationships between services and devices (see center bottom of Figure B.2).

²<http://www.moskitt.org/eng/moskitt0/>

³<http://www.pros.upv.es/labs/projects/pervml>

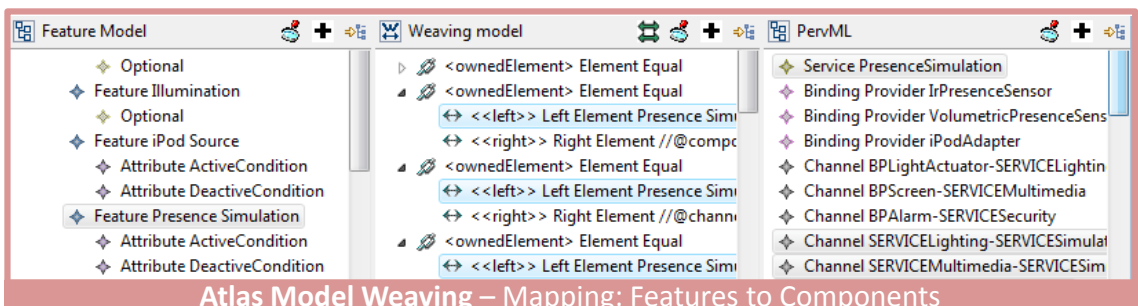
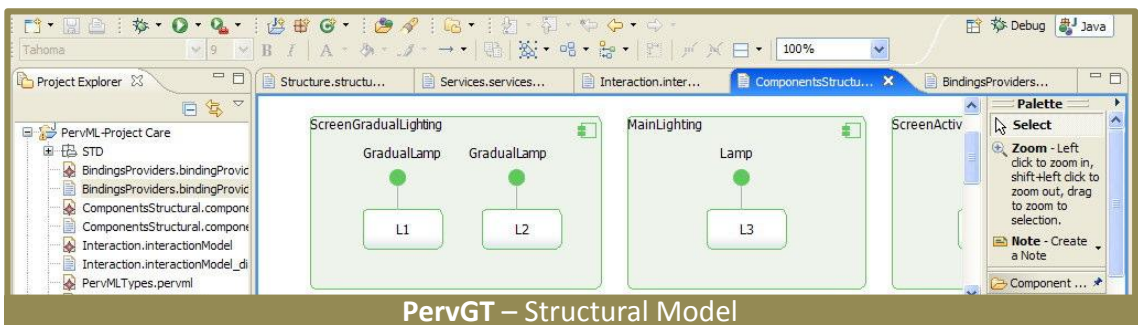
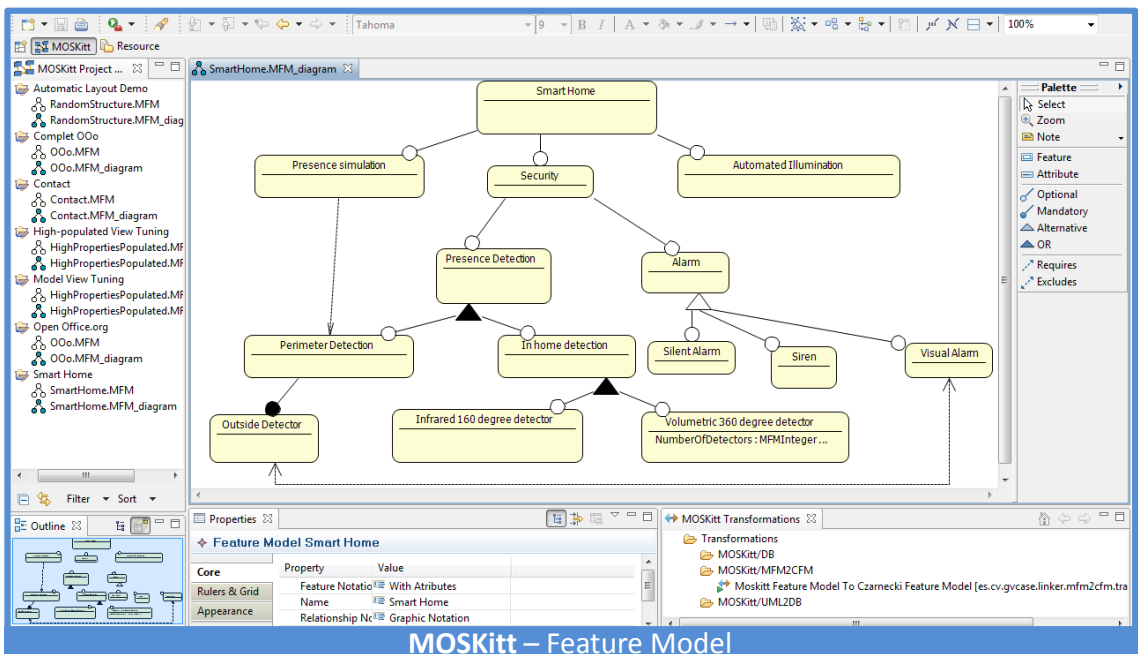
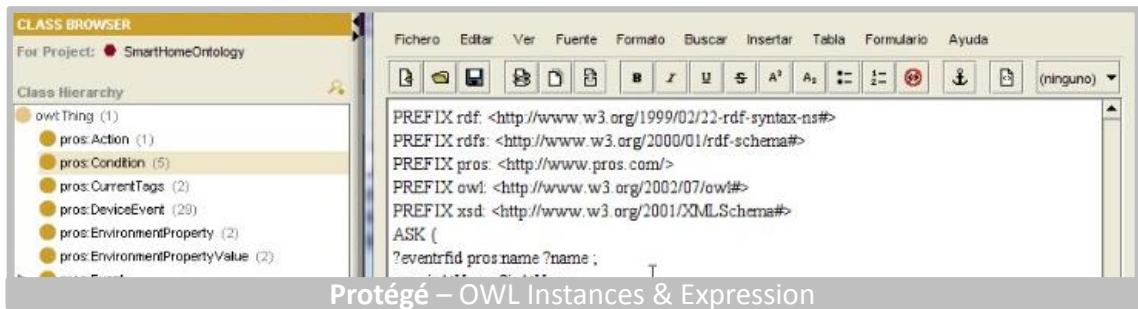


Figure B.2: Screenshots of the Tool Support for Design Time

- **Atlas Model Weaving⁴** (AMW). This tool enables the definition of relations between features and architecture components of PervML. AMW provides a three panels interface. The left panel shows the features of the feature model. The right panel shows the architecture components of the PervML model. Finally, the central panel enables the definition of the relationships by means of the link concept (ElementEqual). Each link denotes a feature (left element) and an architecture component (right element). One to many relationships are defined by composing several links with the same left element (see bottom of Figure B.2) .

The above tools enable an autonomic system engineer to design (1) the system operational environment, (2) context conditions and (3) system features to address the former conditions. Since a given condition can trigger the activation/deactivation of several features, autonomic system engineers define Resolutions to represent the set of changes triggered by a condition.

MOSKitt provides a Resolution Editor which bridges Feature Models and Context conditions (see Figure B.3). First, this editor enables the definition of descriptive information about the resolution such as ID, name, associated self-* property and description. Then, this editor queries the feature model in order to show a list of available features. The autonomic system engineer can assign a feature state (active or inactive) to these features in order to define a partial configuration. A partial configuration is a subset of the system features where each feature has a feature state assigned. This partial configuration describes the effect of the resolution when the context condition is fulfilled. Finally, the context condition of the resolution is set from the SPARQL expressions defined by means of Protege.

By means of the above resolution editor, the autonomic system engineer specifies how the system bind its own variation points, initially when the system is launched to adapt to the current context, as well as during operation to adapt to changes in the context.

⁴<http://www.eclipse.org/gmt/amw/>

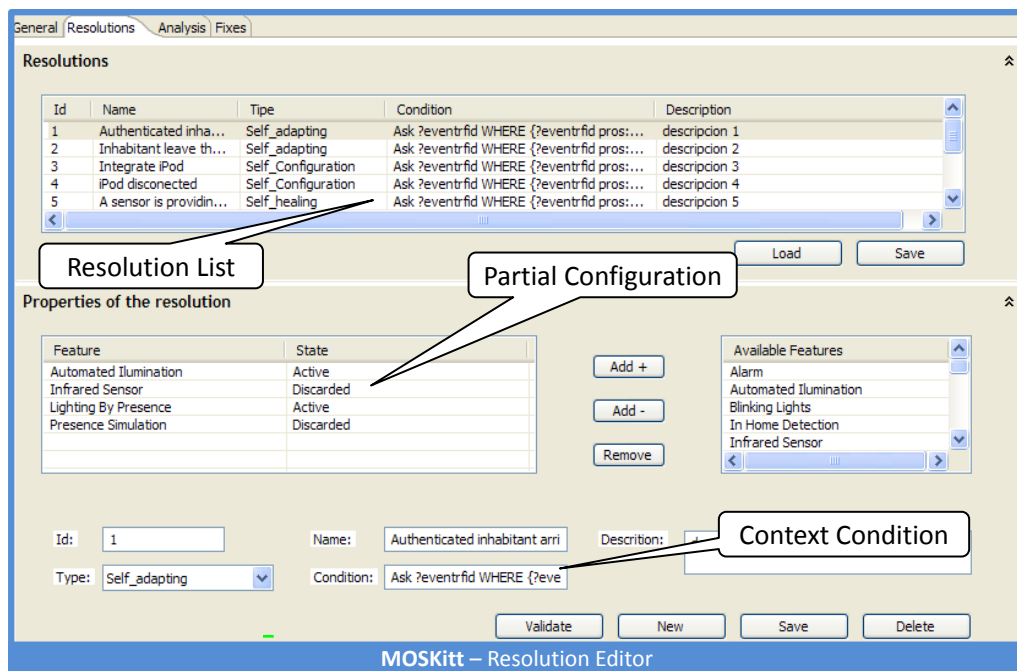


Figure B.3: Screenshot of the Resolution Editor

B.1.1 Support for Reconfiguration Analysis

When an autonomic system engineer defines a Resolution for the activation/deactivation of system features, he/she is expressing the transitions between different system configurations in a declarative manner. We also provide a tool to support the reconfiguration analysis. This tool is based on the analysis operations of the FaMa framework to determine if a particular configuration is valid or invalid according to variability constraints.

First, the reconfiguration analysis tool takes as input the Resolutions and the variability model in order to calculate the resulting possibility space. The possibility space is conformed by all feasible configurations from the fulfilment of context conditions. To calculate the possibility space, the tool takes into account the fulfilment of not only just one context condition but also several context conditions at the same time.

Then, the resulting configurations are validated by means of the FaMa framework. This enables us not only to obtain a valid-invalid tag for each configuration, but also to know the reasons why a particular configuration is invalid. Furthermore,

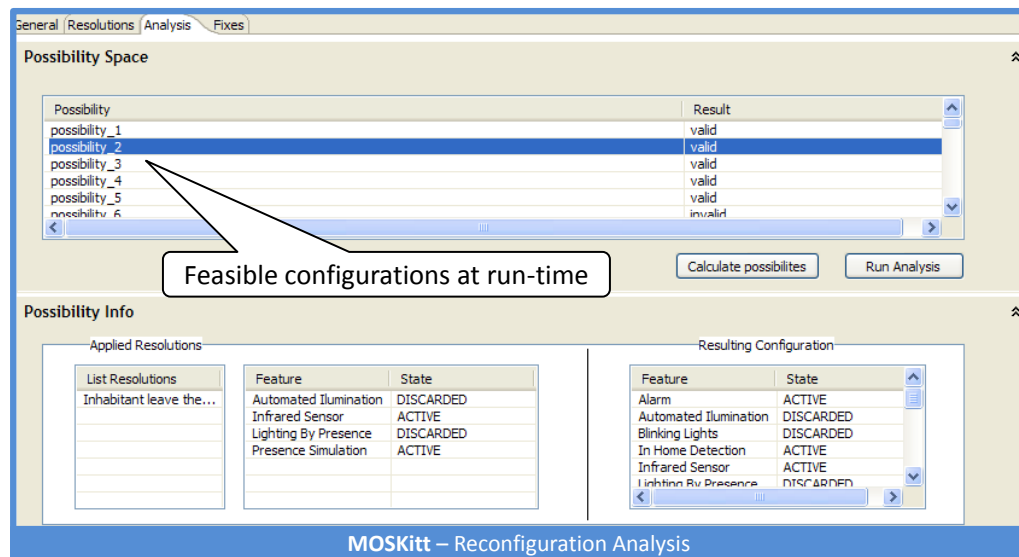


Figure B.4: Screenshot of the Reconfiguration Analysis Tool

the tool also provides an example of resolution path to reach each configuration. This path lists a set of resolutions that can be applied from the initial configuration of the system to the particular configuration.

Given the above information, autonomous system engineers can update either the variability constraints or the resolutions to achieve a specification free of invalid configurations that can be used at run-time.

B.2 Support for Model-based Run-time Reconfigurations

To enable autonomous behaviour, the system must evolve from one configuration to another by itself. Since the reconfiguration in our approach is performed in terms of features, a Model-based Reconfiguration Engine (MoRE) is provided to translate context changes into changes in the activation/deactivation of features. Then, these changes are translated into the reconfiguration actions that modify the system components accordingly.

Currently, MoRE is implemented on top of the OSGi framework. Specifically, we are using the open source OSGi implementation of Proxys (called Equinox). As any OSGi-compliant implementation, Equinox provides a shared execution environment

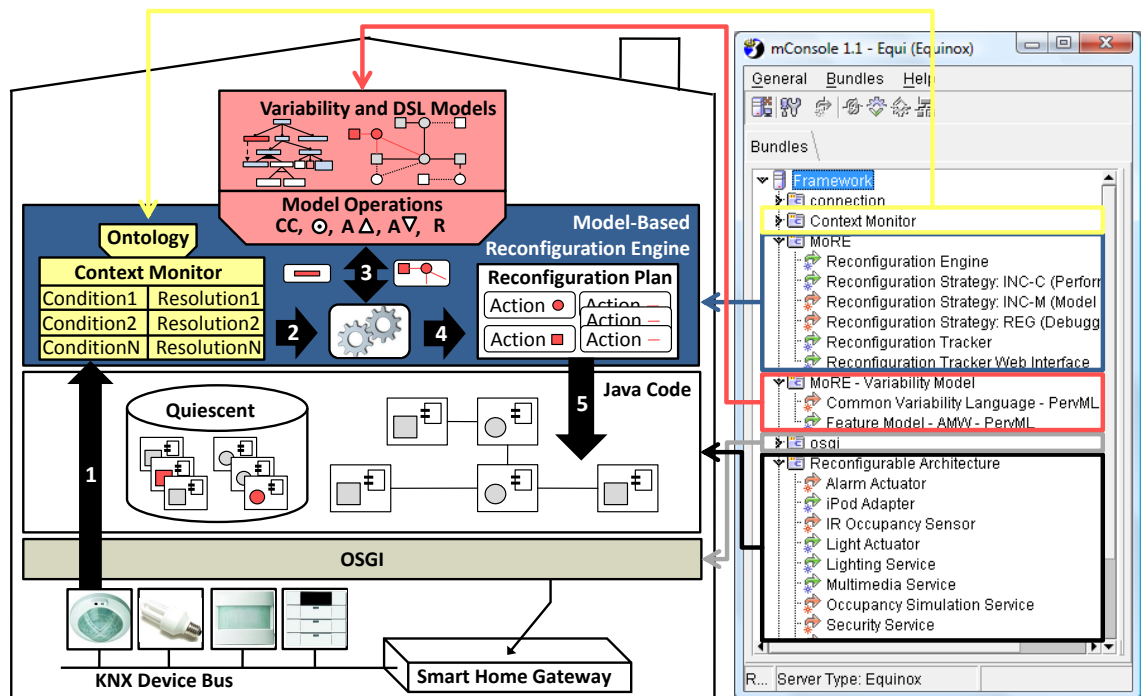


Figure B.5: MoRE implemented as OSGi Bundles.

to install, update, and uninstall components without needing to restart the entire system. In addition, Equinox is also compatible with a set of technology components (KNX, UPnP or EHS) which enables the development of systems in the smart home domain.

System components are known as bundles according to OSGi terminology, and they can register services within the framework service registry. Then, other Bundles can discover registered services and use their functionality.

Figure B.5 shows the management console of Equinox (right), the proposed run-time approach (left) and how our approach is implemented by different OSGi bundles (colour mapping) as follows.

- The **Context Monitor** (yellow) is implemented by means of two bundles. The first bundle implements the main functionality of the context monitor (environment sensing and condition evaluation), and the other bundle stores the OWL context ontology.
- **MoRE** (blue) is implemented by several bundles. The first bundle holds the Reconfiguration engine itself. Other bundles implement the different reconfig-

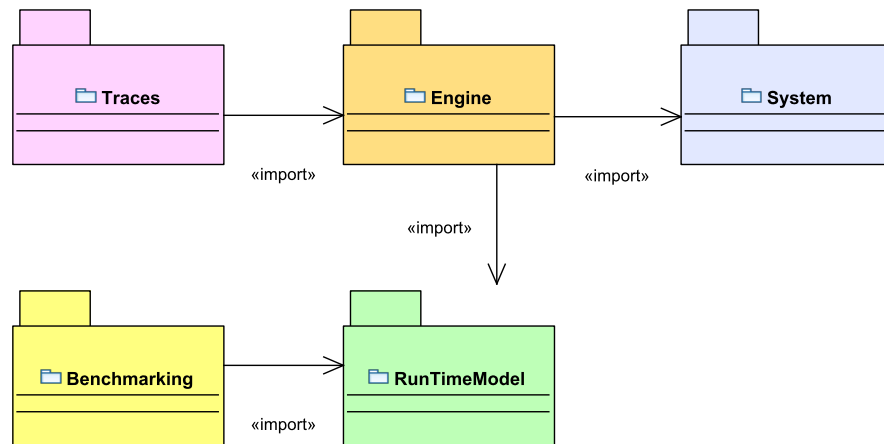


Figure B.6: Package Diagram of MoRE Implementation

uration strategies. Finally, other two bundles hold the Reconfiguration Tracker implementation and its web interface to consul the reconfiguration traces.

- The **Variability Model** (Red) is stored in another bundle. This bundle comprises not only the variability model but also the DSL model and the weaving model.
- The **OSGi** (gray) framework also provides its own bundles such as the one to manage the service registry.
- The **Reconfigurable Architecture** (white) is conformed by the service and device bundles.

To support our approach, some of the above bundles are always started as is the case of the Reconfiguration Engine, the Variability Model, the Reconfiguration Strategy and the Context Monitor. Note that it is possible to install different variability models or reconfiguration strategies as Figure B.5 shows. In addition, other bundles can be optionally started to provide extra functionality as is the case of the Reconfiguration Tracker and its web interface.

Figure B.6 shows the implementation of the above bundles by means of the UML2 Package diagram. The Engine package requires both the RunTimeModel and System packages. The Engine requires these packages to implement the reconfiguration strategy that queries the run-time model (model operations) and modifies the system

architecture (architecture actions). The Traces package requires the Engine package because it records a run of the reconfiguration in term of trace entries. Finally, the Benchmarking package requires the RunTimeModel package because this package dynamically injects instances in the model population to run load tests.

Figure B.7 shows the UML2 class of each one of the above packages. Some of these classes implements the main functionality of MoRE bundles besides the other classes provides interfaces to decouple MoRE of specific platforms or modelling languages as follows.

- *Reconfiguration Engine* Package.
 - **ReconfigurationAction**. This is a common interface that has to be implemented by each reconfiguration action in order to be executed within a Reconfiguration Plan.
 - **ServiceAction**. This class implements the *Decentralized Control System Reconfiguration Pattern*.
 - **ChannelAction**. This class take advantage of the Whiteboard pattern implemented by the OSGi wires to also support the *Decentralized Control System Reconfiguration Pattern*.
 - **ModelAction**. This class takes advantage of the introspection capabilities of EMF Model Query to manipulate the models at run-time.
- *Run-Time Model* Package.
 - **XmiModel**. This class implement common functionality to manipulate XMI-based models such as model save or model element search.
 - **UpdateFeatureAction**. This class implements a reconfiguration action to compose a partial configuration with the current configuration of the variability model.

Finally, although the current implementation runs on top of the desktop version of Equinox, there is also an experimental version running on top of the Android version of Equinox with the aim of bringing MoRE to the domain of mobile devices.

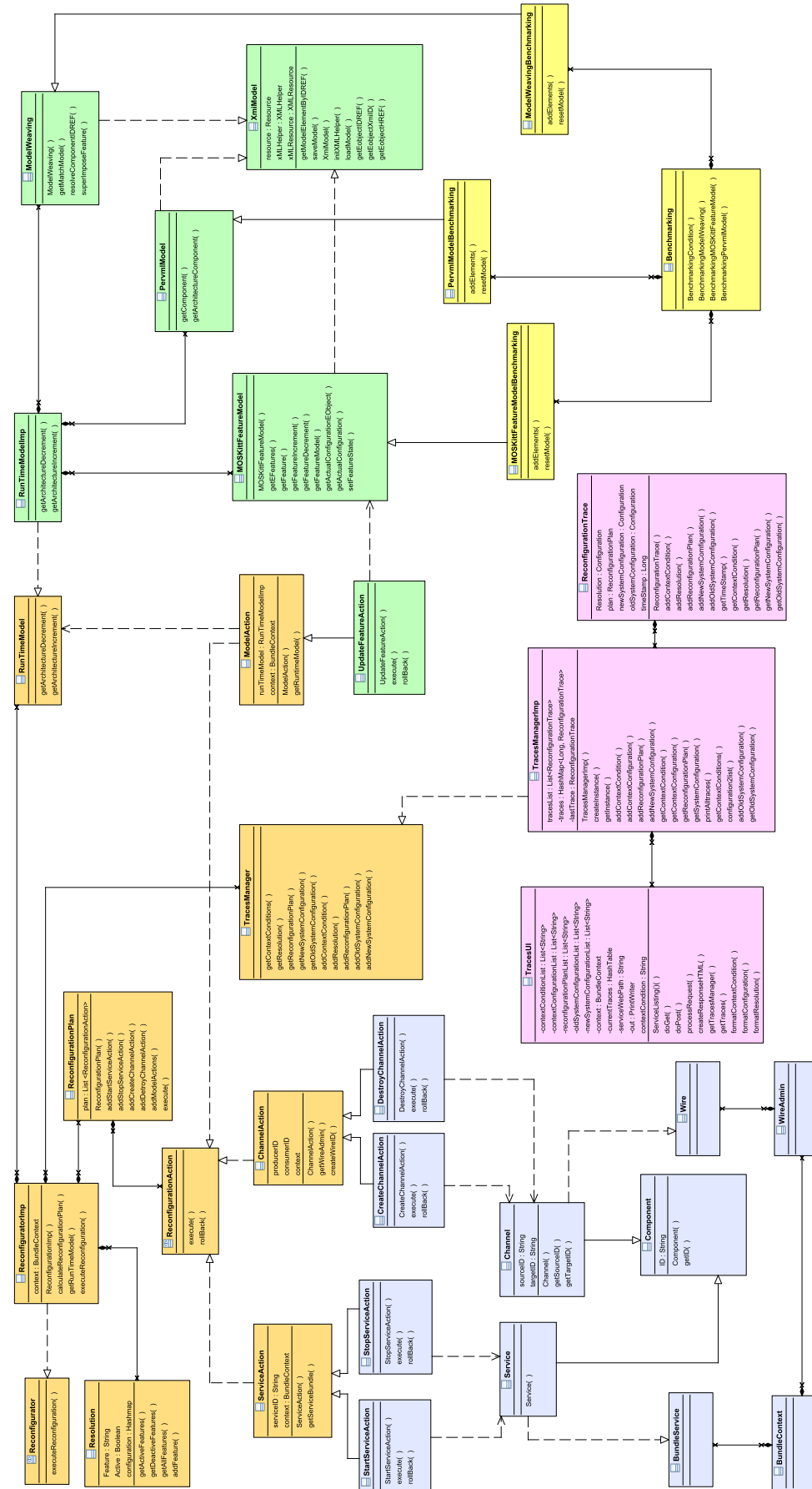


Figure B.7: Class Diagram of MoRE Implementation

BIBLIOGRAPHY

- [1] M. C. Huebscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, 40(3):1–28, August 2008.
- [2] IBM. An architectural blueprint for autonomic computing. Technical report, IBM., 2003.
- [3] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.
- [4] J. Hong, E. Suh, and S. Kim. Context-aware systems: A literature review and classification. *Expert Syst. Appl.*, 36(4):8509–8522, 2009.
- [5] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on*, pages 85–90, 8-9 Dec 1994.
- [6] G. Banavar and A. Bernstein. Software infrastructure and design challenges for ubiquitous computing applications. *Commun. ACM*, 45(12):92–96, 2002.
- [7] J. P. Sousa and D. Garlan. Aura: An architectural framework for user mobility in ubiquitous computing environments. In *In Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, pages 29–43. Kluwer Academic Publishers, 2002.
- [8] V. Bellotti and K. Edwards. Intelligibility and accountability: Human considerations in context-aware systems. *Human-Computer Interaction*, 16:193–212, 2001.

- [9] W. Sitou M. Fahrmaier and B. Spanfelner. Unwanted behavior and its impact on adaptive systems in ubiquitous computing. *ABIS 2006: 14th Workshop on Adaptivity and User Modeling in Interactive Systems*, October 2006.
- [10] P. Horn. *Autonomic computing: IBM's perspective on the state of information technology*, 2001.
- [11] S. Bhola, M. Astley, R. Saccone, and M. Ward. Utility-aware resource allocation in an event processing system. In *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 55–64, Washington, DC, USA, 2006. IEEE Computer Society.
- [12] T. Zenmyo, H. Yoshida, and T. Kimura. A self-healing technique based on encapsulated operation knowledge. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 25–32, June 2006.
- [13] M.L. Littman, N. Ravi, E. Fenson, and R. Howard. Reinforcement learning for autonomic network repair. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 284–285, May 2004.
- [14] H. A. Müller, H. M. Kienle, and U. Stege. Autonomic computing now you see it, now you don't. pages 32–54, 2009.
- [15] S. Kent. Model driven engineering. In *Proceedings of the Third International Conference Integrated Formal Methods (IFM'2002)*, 2002.
- [16] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.
- [17] G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009.
- [18] S. Hallsteinsen, M. Hinchey, Sooyong Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008.

- [19] J. O'Brien, T. Rodden, M. Rouncefield, and J. Hughes. At home with the technology: an ethnographic study of a set-top-box trial. *ACM Trans. Comput.-Hum. Interact.*, 6(3):282–308, 1999.
- [20] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *Software, IEEE*, 15(6):37–45, Nov/Dec 1998.
- [21] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.
- [22] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decis. Support Syst.*, 15(4):251–266, 1995.
- [23] V. Vaishnavi and W. Kuechler. Design research in information systems. <http://www.isworld.org/Researchdesign/drisISworld.htm>, January 2004.
- [24] J. Muñoz and V. Pelechano. Building a software factory for pervasive systems development. In *CAiSE*, pages 342–356, 2005.
- [25] C. Cetina, E. Serral, J. Munoz, and V. Pelechano. Tool support for model driven development of pervasive systems. *mompes*, 0:33–44, 2007.
- [26] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [27] D. F. Bantz, C. Bisdikian, D. Challener, J. P. Karidis, S. Mastrianni, A. Mohindra, D. G. Shea, and M. Vanover. Autonomic personal computing. *IBM Syst. J.*, 42(1):165–176, 2003.
- [28] L. D. Paulson. Computer system, heal thyself. *Computer*, 35(8):20–22, 2002.
- [29] J. A. Mccann and J.S. Crane. Kendra: Internet distribution delivery system. In Society for Computer Simulation International, editor, *In Proceedings of SCS Euromedia*, pages 134–140. IEEE, 1998.

- [30] A Lippman. Video coding for multiple target audiences. In R. L. Stevenson K. Aizawa and Y.-Q. Zhang, editors, *Proceedings of the IS&T/SPIE Conference on Visual Communications and Image Processing*, pages 780–784, 1999.
- [31] A. Ganek and R. J. Friedrich. The road ahead—achieving wide-scale deployment of autonomic technologies. In *Chairing the Town hall meeting at the 3rd IEEE International Conference on Autonomic Computing*, 2006.
- [32] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.
- [33] C. Roblee and G. Cybenko. Implementing large-scale autonomic server monitoring using process query systems. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 123–133, Washington, DC, USA, 2005. IEEE Computer Society.
- [34] J. W. Strickland, V. W. Freeh, Xiaosong Ma, and S. S. Vazhkudai. Governor: Autonomic throttling for aggressive idle resource scavenging. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 64–75, 2005.
- [35] Ji. Xu and J. A. B. Fortes. Towards autonomic virtual applications in the invigo system. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society.
- [36] R. Sterritt, B. Smyth, and M. Bradley. Pact: personal autonomic computing tools. pages 519–527, 2005.
- [37] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung. Self-managing systems: A control theory foundation. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pages 441–448, Washington, DC, USA, 2005. IEEE Computer Society.

- [38] B. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 241–248, New York, NY, USA, 2002. ACM.
- [39] J. R. Pilgrim III W. N. M. J. P. Bigus, D. A. Schlosnagle and Y. Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41:250–371, 2002.
- [40] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, page 3, Washington, DC, USA, 2004. IEEE Computer Society.
- [41] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: A language for specifying security and management policies for distributed systems. Technical report, 2000.
- [42] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. Softw. Eng.*, 25(6):852–869, 1999.
- [43] S. Matsuda K. Broda H. Kamoda, M. Yamaoka and M. Sloman. Policy conflict analysis using free variable tableaux for access control in web services environments. In *Proceedings of the Policy Management for the Web Workshop at the 14th International World Wide Web Conference (WWW)*, 2005.
- [44] A. Gupta. Management of conflicting obligations in self-protecting policy-based systems. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 274–285, Washington, DC, USA, 2005. IEEE Computer Society.
- [45] G. Tesauro and J. O. Kephart. Utility functions in autonomic systems. In *ICAC '04: Proceedings of the First International Conference on Automatic Computing*, pages 70–77, Washington, DC, USA, 2004.

- [46] S. N. Bhatti and G. Knight. Enabling qos adaptation decisions for internet applications. *Comput. Netw.*, 31(7):669–692, 1999.
- [47] J Zhang and B. 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.
- [48] S. Agarwala, Yuan Chen, D. Milojicic, and K. Schwan. Qmon: Qos- and utility-aware monitoring in enterprise systems. In *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 124–133, Washington, DC, USA, 2006. IEEE Computer Society.
- [49] L. Lymberopoulos, E. Lupu, and M. Sloman. An adaptive policy-based framework for network services management. *J. Netw. Syst. Manage.*, 11(3):277–303, 2003.
- [50] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. pages 291–298, 1999.
- [51] D. Agrawal, S. Calo, J. Giles, Kang-Won Lee, and D. Verma. Policy management for networked systems and applications. In *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, pages 455–468, May 2005.
- [52] V. Batra, J. Bhattacharya, H. Chauhan, A. Gupta, M. Mohania, and U. Sharma. Policy driven data administration. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 0:0220, 2002.
- [53] H. Lutfiyya, G. Molenkamp, M. Katchabaw, and M. A. Bauer. Issues in managing soft qos requirements in distributed systems using a policy-based framework. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 185–201, London, UK, 2001. Springer-Verlag.
- [54] A. Ponnappan, L. Yang, R. Pillai, and P. Braun. A policy based qos management system for the intserv/diffserv based internet. In *POLICY '02: Proceed-*

- ings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, page 159, Washington, DC, USA, 2002. IEEE Computer Society.
- [55] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.
- [56] D. Garlan, B. Schmerl, and J Chang. Using gauges for architecture-based monitoring and adaptation. In *In Working Conference on Complex and Dynamic Systems Architecture , Brisbane, Australia.*, 2001.
- [57] A. A. Bougaev. Pattern recognition based tools enabling autonomic computing. pages 313–314, 2005.
- [58] A. Salahshour E. Manoel, M. J. Nielsen and S. Sampath. *Problem Determination Using Self-Managing Autonomic Technology*. IBM Redbooks, 2005.
- [59] P. Shivam, S. Babu, and J. S. Chase. Learning application models for utility resource planning. In *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 255–264, Washington, DC, USA, 2006. IEEE Computer Society.
- [60] T. Osogami, M. Harchol-Balter, and A. Scheller-Wolf. Analysis of cycle stealing with switching times and thresholds. *Perform. Eval.*, 61(4):347–369, 2005.
- [61] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu. Power-aware qos management in web servers. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 63, Washington, DC, USA, 2003. IEEE Computer Society.
- [62] R. S. Sutton and A. G Barto. *Reinforcement learning: An Introduction*. MIT Press, 1998.
- [63] E. Fenson and R. Howard. Reinforcement learning for autonomic network repair. In *ICAC '04: Proceedings of the First International Conference on*

- Autonomic Computing*, pages 284–285, Washington, DC, USA, 2004. IEEE Computer Society.
- [64] J. Dowling, R. Cunningham, E. Curran, and V. Cahill. Building autonomic systems using collaborative reinforcement learning. volume 21, pages 231–238, New York, NY, USA, 2006. Cambridge University Press.
- [65] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 65–73, June 2006.
- [66] S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *J. Mach. Learn. Res.*, 7:877–917, 2006.
- [67] H. Guo. A bayesian approach for autonomic algorithm selection. In *Proceedings of the IJCAI workshop on AI and autonomic computing: developing a research agenda for selfmanaging computer systems*, 2003.
- [68] T. Nguyen M. Littman and H. Hirsh. A model of cost-sensitive fault mediation. In *Proceedings of the IJCAI workshop on AI and autonomic computing: developing a research agenda for self-managing computer systems*, 2003.
- [69] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.
- [70] A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, and G. Karsai. Generative programming via graph transformations in the model-driven architecture. In *In OOPSLA 2002 Workshop in Generative Techniques in the context of Model Driven Architecture*, 2002.
- [71] S. J. Mellor and A. N. Clark and T. Futagami. Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, 20(5):14–18, 2003.
- [72] A. Sernadas, C. Sernadas, and H.-D. 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.

- [73] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL: a language for object-oriented specification of information systems. *ACM Trans. Inf. Syst.*, 14(2):175–211, 1996.
- [74] M. Rohs and J. 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.
- [75] Object Management Group. Unified Modeling Language: Superstructure version 2.1.1. OMG Specification, February 2007.
- [76] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [77] I. Kurtev. *Adaptability of Model Transformations*. phdthesis, IPA, 2005. ISBN 90-365-2184-X.
- [78] J. Bézivin, N. Farcet, J. Jézéquel, B. Langlois, and D. Pollet. Reflective model driven engineering. pages 175–189. Springer, 2003.
- [79] J. Bézivin. In search of a basic principle for model driven engineering. *UP-GRADE*, 1:15–24, 2004.
- [80] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.
- [81] J. Miller and J. Mukerji. Mda guide version 1.0.1, 2003. Letzte Änderung am 12. Jun. 2003, besucht am 15. Mai 2008.
- [82] R. Balzer. A 15 year perspective on automatic programming. *IEEE Trans. Softw. Eng.*, 11(11):1257–1268, 1985.
- [83] K. Czarnecki. *Generative Programming. Principles and Techniques of Software Engineering Based on Automated Conguration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, October 1998.

- [84] U. W. Eisenecker. Generative programming (gp) with c++. In *JMLC '97: Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, pages 351–365, London, UK, 1997. Springer-Verlag.
- [85] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. volume 35, pages 26–36, New York, NY, USA, 2000. ACM.
- [86] S. Kelly J.P. Tolvanen and M.C. Consulting. Modelling languages for product families a method engineering approach. In *1st OOPSLA Workshop on Domain-specific Visual Languages*, 2001.
- [87] Io Ns and M. Simos. *Organization Domain Modeling and OO Analysis and Design: Distinctions, Integration, New Directions*. 1997.
- [88] R. Esser and J.W. Janneck. A framework for defining domain-specific visual languages. In *OOPSLA 2001 Workshop on Domain Specific Visual Languages*, 2001.
- [89] 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.
- [90] D.L. Parnas. On the design and development of program families. *Software Engineering, IEEE Transactions on*, SE-2(1):1–9, March 1976.
- [91] 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.
- [92] F. van der Linden. *Lecture Notes in Computer Science*, volume 2290. Springer, Bilbao, Spain, October 3-5, 2001 2002.
- [93] P. Donohoe. Number ISBN 0-7923-7940-3. Denver, Colorado, USA, August 28-31.

- [94] D. Batory, J. Neal Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:2004, 2003.
- [95] L. Northrop. SEI's Software Product Line Tenets. *IEEE Software*, 19(4):32–40, July/August 2002.
- [96] G. Chastek and J.D. McGregor. Guidelines for developing a product line production plan. Technical report, CMU/SEI, June 2002.
- [97] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press, New York, NY, USA, 2000.
- [98] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [99] J. D. McGregor, L. M. Northrop, S. Jarrad, and K. Pohl. Guest editors' introduction: Initiating software product lines. *IEEE Software*, 19(4):24–27, 2002.
- [100] Software product-family engineering. In Frank van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*. Springer, 2004.
- [101] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 1990.
- [102] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM.
- [103] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, 2003.
- [104] R. Adler, D. Schneider, and M. Trapp. Development of safe and reliable embedded systems using dynamic adaptation. *Workshop on Model-Driven Software Adaptation, M-ADAPT 2007 at ECOOP*, pages 9–14, 2007.

- [105] H. Gomaa and G.A. Farrukh. Methods and tools for the automated configuration of distributed applications from reusable software architectures and components. *Software, IEE Proceedings* -, 146(6):277–290, Dec 1999.
- [106] J. Lee and K.C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Software Product Line Conference, 2006 10th International*, 2006.
- [107] K. Kim. Dynamic reconfiguration of architecture based on component interactions. Master’s thesis, Dept. of CSE, POSTECH, 2006.
- [108] 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.
- [109] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating product-line variant selection for mobile devices. *Software Product Line Conference, 11th International*, pages 129–140, 10-14 Sept. 2007.
- [110] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 1994.
- [111] P. Trinidad, A. Ruiz-Cortés, and J. Peña. Mapping feature models onto component models to build dynamic software product lines. *International Workshop on Dynamic Software Product Line*, 2007.
- [112] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *MoDELS ’08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 782–796, Berlin, Heidelberg, 2008. Springer-Verlag.
- [113] S. Hallsteinsen, S. Jiang, and R. Sanders. Dynamic software product lines in service oriented computing. In *3rd International Workshop on Dynamic Software Product*, 2009.

- [114] C. Parra, X. Blanc, and L. Duchien. Context Awareness for Dynamic Service-Oriented Product Lines. *Software Product Line Conference, 2009. SPLC 2009. 13th International*, 24-28 August. 2009.
- [115] OW2 consortium. Frascati project. <http://frascati.ow2.org>.
- [116] R. Rouvoy, D. Conan, and L. Seinturier. Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online*, 9(6):1, 2008.
- [117] P. Istoan, G. Nain, G. Perrouin, and J.M. Jézéquel. Dynamic software product lines for service-based systems. *Computer and Information Technology, International Conference on*, 2:193–198, 2009.
- [118] P. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of MODELS/UML 2005*. Springer, 2005.
- [119] D. Berry, B. Cheng, and J. Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *In 11th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ)*, 2005.
- [120] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *Proceedings of the Third Software Product Line Conference 2004*, pages 266–282. Springer, LNCS 3154, 2004.
- [121] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, Gø. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008*, pages 139–148, Limerick, Ireland, 2008.
- [122] M. Dean and G. Schreiber. OWL web ontology language reference. W3C recommendation, W3C, February 2004.
- [123] Y. Liu, Muhammad A. Babar, and I. Gorton. Middleware architecture evaluation for dependable self-managing systems. In *QoSA '08: Proceedings of the 4th International Conference on Quality of Software-Architectures*, pages 189–204, Berlin, Heidelberg, 2008. Springer-Verlag.

- [124] D. Marples and P. Kriens. The open services gateway initiative: an introductory overview. *Communications Magazine, IEEE*, (12):110–114, Dec 2001.
- [125] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, 2009.
- [126] C. Cetina, Ø. Haugen, X. Zhang, F. Fleurey, and V. Pelechano. Strategies for variability transformation at run-time. In *Software Product Line Conference, 2009. SPLC '09. 13th International*, August 2009.
- [127] R. C. Linger, B. I. Witt, and H. D. Mills. *Structured Programming; Theory and Practice the Systems Programming Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979.
- [128] H. A. Schmid. Creating applications from components: A manufacturing framework design. *IEEE Softw.*, 13(6):67–75, 1996.
- [129] K. Czarnecki. Overview of generative software development. In *Unconventional Programming Paradigms, International Workshop UPP*, pages 326–341, 2004.
- [130] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.
- [131] D. Schmidt, A. Nechypurenko, and E. Wuchner. Workshop mdd for software product-lines: Fact or fiction. *International Conference on Model Driven Engineering Languages and Systems*, 2005.
- [132] G. Botterweck, I. Groher, A. Polzer, C. Schwanninger, S. Thiel, and M. Voelter. International workshop on model-driven approaches in software product line. *13th International Software Product Line Conference*, 2009.
- [133] P. Schobbens, J.C. Trigaux P. Heymans, and Y. Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2):456–479, 2007.

- [134] J. Muñoz, V. Pelechano, and C. Cetina. Implementing a pervasive meeting room: A model driven approach. In *IWUC*, pages 13–20, 2006.
- [135] J. Muñoz Ferrara. *Model Driven Development of Pervasive Systems. Building a Software Factory*. Tesis doctoral en informática, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2008.
- [136] D. Del Fabro. *Metadata management using model weaving and model transformation*. PhD thesis, University of Nantes, September 2007.
- [137] M. Didonet Del Fabro, J. Bézivin, and P. Valduriez. Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany*, 2006.
- [138] E. Serral, P. Valderas, and V. Pelechano. A model driven development method for developing context-aware pervasive systems. In *UIC '08: Proceedings of the 5th international conference on Ubiquitous Intelligence and Computing*, Berlin, 2008.
- [139] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Using feature models for developing self-configuring smart homes. Fifth International Conference on Autonomic and Autonomous Systems (ICAS 2009), April 2009.
- [140] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 2010.
- [141] 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.
- [142] P. Trinidad and A. Ruiz-Cortés. Abductive reasoning and automated analysis of feature models: How are they connected? (submitted). In *Proceeding of the Third International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2009.
- [143] J. Bayer, S. Gerard, Ø. Haugen, J. Mansell, B. Møller-Pedersen, J. Oldevik, J. Thibault P. Tessier and, and T. Widen. Consolidated product line variability

- modeling. in software product lines. In Springer, editor, *Research Issues in Engineering and Management*, 2006.
- [144] ITEA project ip02009. Families: Families, in eureka s! 2023 programme, 2004.
- [145] G. Blair, N. Bencomo, and France R. Models@ run.time. *Computer*, pages 22–27, October 2009.
- [146] N. Bencomo, G. Blair, and R. France. 3rd workshop on models@run.time at models 2008 (proceedings). Technical Report COMP-005-2008 Lancaster University.
- [147] H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *4th Working IEEE / IFIP Conference on Software Architecture*, pages 79–88, 2004.
- [148] H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. In *Software Product-Family Engineering, 5th International Workshop*, pages 435–444, 2003.
- [149] H. Gomaa. Designing concurrent, distributed, and real-time applications with uml. In *28th International Conference on Software Engineering ICSE*, pages 1059–1060, 2006.
- [150] T. Gu, H. Keng Pung, and Da Qing Zhang. Toward an osgi-based infrastructure for context-aware applications. *IEEE Pervasive Computing*, 3(4):66–74, 2004.
- [151] D. Zhang, X. Hang Wang, and K. Hackbarth. Osgi based service infrastructure for context aware automotive telematics. pages 81–88, 2004.
- [152] K. Czarnecki. Mapping features to models: A template approach based on superimposed variants. In *GPCEŠ05, volume 3676 of LNCS*, pages 422–437, 2005.

- [153] D. Beuche. Modeling and building software product lines with pure::variants. In *Software Product Line Conference, 2009. SPLC '09. 13th International*, August 2009.
- [154] C.W. Krueger, D. Churchett, and R. Buhrdorf. Homeaway's transition to software product line practice: Engineering and business results in 60 days. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 297–306, Sept. 2008.
- [155] C.W. Krueger. The biglever software gears unified software product line engineering framework. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 353–353, Sept. 2008.
- [156] J. Martínez, C. López, A. Aldazabal, J. Mansell, and M. del Hierro. Plum (product line unified modeller). In *Software Product Line Conference, 2009. SPLC '09. 13th International*, August 2009.
- [157] A. J. Ramirez, D. B. Knoester, B. H.C. Cheng, and P. K. McKinley. Applying genetic algorithms to decision making in autonomic computing systems. pages 97–106, 2009.
- [158] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.
- [159] Mof: Mof 2.0 (meta object facility). <http://www.omg.org/spec/MOF/2.0/>, 2006.
- [160] C. Cetina, J. Fons, and V. Pelechano. Applying Software Product Lines to Build Autonomic Pervasive Systems. *Software Product Line Conference, 2008. SPLC 2008. 12th International*, 8-12 Sept. 2008.
- [161] Eclipse Modeling Framework (EMF) - Model Query website. <http://www.eclipse.org/modeling/emf/?project=query>.

- [162] F. Jouault and I. Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference, LNCS 3844*, pages 128–138. Springer, 2006. ISBN=0302-9743.
- [163] Eclipse MOFscript web-site. MODELWARE (IST Project 511731). <http://www.eclipse.org/gmt/mofscript/>.
- [164] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December, 2006.
- [165] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.
- [166] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, and A. Ruiz Cortés. Automated diagnosis of product-line configuration errors in feature models. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, 2008.
- [167] F. Loesch and E. Ploedereder. Optimization of variability in software product lines. *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 151–162, 10-14 Sept. 2007.
- [168] P. Giner, C. Cetina, J. Fons, and V. Pelechano. Building Self-adaptive services for Ambient Assisted Living. In *International Workshop of Ambient Assisted Living IWAAL09, LNCS*, 2009.
- [169] S. M. Yacoub and H. H. Ammar. A methodology for architecture-level reliability risk analysis. *IEEE Trans. Softw. Eng.*, 28(6):529–547, 2002.
- [170] B. González-Baixauli, M. Laguna, and Y. Crespo. Product line requirements based on goals, features and use cases. *International Workshop on Requirements Reuse in System Family Engineering*, pages 4–6, 2004.
- [171] S. Jarzabek, B. Yang, and S. Yoeun. Addressing quality attributes in domain analysis for product lines. *Software, IEE*, 153(2):61–73, April 2006.

- [172] H. Zhang, S. Jarzabek, and B. Yang. Quality prediction and assessment for product lines. page 1031. 2003.
- [173] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *SPLC*, pages 197–213, 2004.
- [174] E. Niemelä and A. Immonen. Capturing quality requirements of product family architecture. *Inf. Softw. Technol.*, 49(11-12):1107–1120, 2007.
- [175] A. Immonen. A method for predicting reliability and availability at the architecture level. In *Software Product Lines*, pages 373–422. 2006.
- [176] L. Etxeberria and G. Sagardui. Variability driven quality evaluation in software product lines. In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, pages 243–252, Washington, DC, USA, 2008. IEEE Computer Society.
- [177] L. Williams and C. Smith. Pasa a method for the performance assessment of software architectures. In *Proceedings of the Third International Workshop on Software and Performance (WOSP2002)*, pages 179–189. ACM Press, 2002.
- [178] E. Folmer, J. van Gorp, and J. Bosch. Scenario-based assessment of software architecture usability. In *Proceedings of Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction*, pages 61–68, 2003.
- [179] R.L. Rosnow and R. Rosenthal. People studying people: Artifacts and ethics in behavioral research. *W.H. Freeman and Company*, 1997.
- [180] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at run-time: The case of smart homes. *Computer*, pages 46–52, October 2009.
- [181] M. J. Weal, D. Cruickshank, D. T. Michaelides, K. Howland, and G. Fitzpatrick. Supporting domain experts in creating pervasive experiences. In *Pervasive Computing and Communications (PerCom)*, pages 108–113, 2007.

- [182] A. K. Dey. Modeling and intelligibility in ambient environments. *Journal of Ambient Intelligence and Smart Environments (JAISE)*, 1(1):57–62, January 2009.
- [183] 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.
- [184] W. Wayt Gibbs. Considerate computing. *Scientific American*, 292(1):54–61, 2004.
- [185] H. 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.
- [186] C. Cetina, P. Trinidad, V . Pelechano, and A. Ruiz-Cortés. An architectural discussion on dspl. *2nd International Workshop on Dynamic Software Product Line (DSPL08)*, 2008.
- [187] Models@run.time workshop in conjunction with with models conference. <http://www.comp.lancs.ac.uk/bencomo/MRT/>.
- [188] Dynamic software product lines workshop in conjunction with with splc conference. <http://www.lero.ie/dspl2009>.
- [189] W.F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998.
- [190] C. Lueg. On the gap between vision and feasibility. In *Pervasive '02: Proceedings of the First International Conference on Pervasive Computing*, pages 45–57, London, UK, 2002. Springer-Verlag.
- [191] A. Schmidt and L. Terrenghi. Methods and guidelines for the design and development of domestic ubiquitous computing applications. *percom*, 00:97–107, 2007.

ABOUT



Carlos Cetina is a research fellow at the Centre of Software Production Methods, and he is currently teaching Databases (Technical University of Valencia) and Model Driven Software Development (Ministry of Infrastructure and Transport of Valencia). His research interests are related to Model Driven Development (MDD), Software Product Lines (SPL), Pervasive Systems and Autonomic Computing. He is also interested in cutting edge Smart Home technologies.

From 2006 to 2007, he has been applying MDD to Pervasive Systems. Specifically, he has been working on (1) a Domain Specific language for the specification of Smart Homes (PervML), and (2) a tool (PervGT) for automatic code generation from PervML models to the final system implementation. See more information of this work at <http://www.pros.upv.es/labs/projects/pervml>.

From 2007 to nowadays, he has been combining both MDD and SPL to achieve autonomic computing. His research shows that variability models at run-time can assist a system to determine the steps that are necessary to reconfigure itself. In particular, he argues that a system can activate/deactivate its own features dynamically at run-time according to the fulfilment of context conditions. See more information of this work at <http://www.autonomic-homes.com/>.

Until Carlos Cetina completely ran out of free time, he used to play rowing with the team of the Technical University of Valencia. Now, he is becoming engrossed with the complex and serious tasks of showing both scalextric and classic videogames to his young nephews. See more information about Carlos Cetina at <http://www.carloscetina.com/>.