Optimizing Mobile Applications by Exploiting Variability Models at Runtime

Gustavo García Pascual

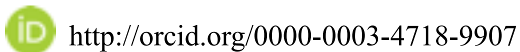
Supervised by
Prof. Dr. Lidia Fuentes Fernández
Dr. Mónica Pinto Alarcón

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga

December 2018

UNIVERSIDAD
DE MÁLAGA

AUTOR: Gustavo García Pascual


http://orcid.org/0000-0003-4718-9907

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga

Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional:
http://creativecommons.org/licenses/by-nc-nd/4.0/legalcode
Cualquier parte de esta obra se puede reproducir sin autorización
pero con el reconocimiento y atribución de los autores.
No se puede hacer uso comercial de la obra y no se puede alterar, transformar o hacer obras derivadas.

Esta Tesis Doctoral está depositada en el Repositorio Institucional de la Universidad de Málaga (RIUMA): riuma.uma.es

UNIVERSIDAD DE MÁLAGA
DEPARTAMENTO DE LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

La Dra. Doña Lidia Fuentes Fernández, Catedrática de Universidad, perteneciente al Área de Telemática, y la Dra. Doña Mónica Pinto Alarcón, Titular de Universidad, perteneciente al Área de Lenguajes y Sistemas Informáticos, de la E.T.S. de Ingeniería Informática de la Universidad de Málaga,

certifican que Don Gustavo García Pascual, Ingeniero de Telecomunicación, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo nuestra dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulado:

Optimizing Mobile Applications by Exploiting Variability Models at Runtime

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Málaga, Diciembre de 2018

Fdo.: Lidia Fuentes Fernández	Fdo.: Mónica Pinto Alarcón
Catedrática de Universidad	Titular de Universidad
Área de Telemática	Área de Lenguajes y Sistemas Informáticos

Acknowledgements

This thesis has been partially funded by the European project INTER-TRUST FP7-317731, the Spanish Government projects MAVI TIN2012-34840 and HADAS TIN2015-64841-R, the Andalusian Government projects FamiWare P09-TIC-5231 and MAGIC P12-TIC1814 and the University of Málaga. It has also been partially funded by the Austrian Science Fund (FWF) project P25289-N15 and by the Lise Meitner Fellowship M1421-N15.

Special Acknowledgements

Este manuscrito representa el final de una etapa. Una etapa que termina dejándome regalos de un valor incalculable: un magnífico grupo de amigos y un puñado de momentos inolvidables vividos a vuestro lado. Por eso, estas líneas son para vosotros.

Para mis maravillosos compañeros del laboratorio 3.3.3, con los que he tenido y sigo teniendo la suerte de compartir y disfrutar todos estos años. Muchas gracias, Josemi, porque entre bachata y bachata siempre has estado ahí dispuesto a ayudar a los demás sin esperar nunca nada a cambio.

Para mis supervisoras, Lidia y Mónica. Sin vuestra ayuda, vuestros consejos, vuestras ideas y vuestra paciencia nada de esto habría sido posible.

Para mi familia, que siempre me ha acompañado y me ha apoyado incondicionalmente.

Para Andrea, por estar siempre conmigo y cuidarme, en los buenos y en los malos momentos. Siempre que lo he necesitado me has ofrecido tu mano y me has guiado, ayudándome a seguir hacia adelante.

Y para Nora, el mejor regalo que me ha dado la vida, y que algún día leerá con curiosidad estas líneas.

Table of contents

Table of contents **ix**

List of figures **xi**

List of tables **xiii**

Nomenclature **xiii**

I Introduction **1**

1 Motivation and Challenges **5**

2 Background **9**

2.1 Dynamic Software Product Lines (DSPLs) 9

2.2 Genetic Algorithms 13

3 Related Work **15**

3.1 Self-adaptation approaches 15

3.2 Optimization algorithms for FMs 21

4 Approach Overview **25**

5 Discussion of Results **31**

6 Conclusions and Future Work **35**

II Publications Composing the PhD Thesis **37**

7 Self-adaptation of Mobile Systems Driven by the Common Variability Language **39**

8 Applying Multiobjective Evolutionary Algorithms to Dynamic Software Product Lines for Reconfiguring Mobile Applications **41**

9 Automatic Analysis of Software Architectures with Variability **43**

10 Run-Time Adaptation of Mobile Applications using Genetic Algorithms **45**

III Appendices **47**

Appendix A Resumen en español **49**

A.1 Antecedentes 50

A.1.1 Líneas de Producto Software Dinámicas 50

A.1.2 Algoritmos genéticos 55

A.2 Retos 56

A.3 Resumen de la propuesta 57

A.4 Contribuciones 62

A.5 Conclusiones y trabajo futuro 64

References **67**

List of figures

2.1 Feature Model Example 11
2.2 CVL Approach 13

4.1 Approach overview 27

A.1 Ejemplo de Modelo de Características 52
A.2 Enfoque CVL 54
A.3 Vista general de la propuesta 61

List of tables

3.1 Self-adaptation approaches 16
3.2 Optimization algorithms for FMs 22

Part I

Introduction

Mobile applications run in an environment where the context is continuously changing. In this thesis we explore the applicability of the Dynamic Software Product Lines (DSPL) [Hallsteinsen et al., 2008] paradigm to develop applications for mobile devices that can be reconfigured at runtime. The DSPL paradigm enables the generation of software capable of adapting to changes in the environment by modelling the elements that can be reconfigured as dynamic variation points. Therefore, it is possible to generate, at runtime, different variants of the DSPL. Our approach covers both the design of the DSPL and the development of the runtime configuration mechanisms. Firstly, it allows software architects to specify DSPLs for mobile devices. Secondly, it provides as a middleware layer with services for monitoring the context and reconfiguring mobile applications. In existing DSPLs for mobile devices, dynamic reconfiguration is typically addressed by generating at design time the configurations that will be deployed at runtime [Rosenmüller et al., 2011, Rouvoy et al., 2009, Shen et al., 2011, White et al., 2007], as well as the differences between pairs of configurations and the conditions to adapt the system from one configuration to another one. All this information is loaded into the mobile device, which results on the deployment of sub-optimal application configurations. In order to cope with this limitation, we propose to extend existing evolutionary algorithms in the context of SPLs to generate, at runtime, variants of the application that are tailored to the current execution context and are very close to the optimal one regarding different criteria. Being this an NP-hard problem [White et al., 2009], it is not possible to use exact techniques typically applied in SPLs, such as Constraint Satisfaction Problems (CSP) [Trinidad et al., 2007], to generate at runtime the optimal solution for the current execution context. However, as part of our work, we demonstrate that configurations generated using our algorithms are close to the optimal one and that these algorithms are efficient enough to be used on mobile devices.

This PhD thesis is presented as a compilation of a set of publications. To this end, Part II of this dissertation includes four publications that shape the main results of this thesis. Two of these articles have been published in journals indexed in the first quartile (Q1). The rest of them have been published in international conferences, being one of them included in the A category in the CORE Conference Ranking.

Before enumerating these publications, this part of the thesis is organized as follows. Chapter 1 presents the motivation and challenges of this thesis. Chapter 2 provides the backgrounds necessary to understand our work, and Chapter 3 discusses the related work. In Chapter 4, we show an overview to the processes of modelling reconfigurable mobile applications and reconfiguring them at runtime to adapt to the context changes. Chapter 5 details the main contributions of this work, specifying the publications where they are presented. Finally, Chapter 6 describes the conclusions of this work and suggests how to deep

in the research presented in this thesis as part of future work.

The publications included as part of this PhD thesis are the following:

1. Self-adaptation of Mobile Systems Driven by the Common Variability Language [Pascual et al., 2015b].
   G. G. Pascual, M. Pinto and L. Fuentes.
   Future Generation Computer Systems (2015).
   JCR Q1

2. Applying Multi-Objective Evolutionary Algorithms to Dynamic Software Product Lines for Reconfiguring Mobile Applications [Pascual et al., 2015a].
   G. G. Pascual, R. López-Herrejón, M. Pinto, L. Fuentes and A. Egyed
   Journal of Systems and Software (2015).
   JCR Q1

3. Automatic Analysis of Software Architectures with Variability [Pascual et al., 2013a].
   G. G. Pascual, M. Pinto and L. Fuentes.
   13th International Conference on Software Reuse (ICSR 2013).
   CORE A

4. Run-Time Adaptation of Mobile Applications using Genetic Algorithms [Pascual et al., 2013b].
   G. G. Pascual, M. Pinto and L. Fuentes.
   8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2013).

Chapter 1

Motivation and Challenges

Mobile devices are a central part in our lives. Nowadays, smartphones are very powerful devices that can run complex applications which are used everywhere and at any time. However, mobile computation is not limited to mobile phones. For instance, wearable devices such as smartwatches (e.g. Android Wear) and glasses (e.g. Google Glass) are increasingly popular, and these devices also run applications which are strongly related to their context. These applications could greatly benefit from applying reconfiguration mechanisms too. Developing applications that adapt to the context where these devices run (e.g. location, available resources...) is fundamental to satisfy the user requirements, as illustrated in the following example. Let us suppose a tourist that is using its mobile device to enrich his tourism experience. He is listening to his favourite music using a streaming service, and an application is notifying him about those points of interest that are in the nearby and those interesting places he is heading to (e.g. restaurants, shops, etc.). Another application is automatically sharing with his friends all the photos he is taking. The user wants the battery to last until he is back to the hotel, where it can be recharged. However, during his route, it is detected that the battery level is too low and it will be depleted soon. Nevertheless, it would be possible to make the battery last longer while satisfying the user requirements by reconfiguring the mobile applications. For instance, the music streaming application could reduce the quality of the stream, or play only locally cached music; the tourist information application could use WiFi and mobile networks for localization instead of GPS satellites; and the photo sharing application could reduce the size of the photos before uploading them, or upload just the most relevant photos. Therefore, in this case, reconfiguring the mobile applications would help to continue satisfying the user requirements, reducing at the same time the battery consumption.

From an engineering point of view, modelling reconfigurable mobile applications and deploying the optimal configurations at runtime is a complex task. Resources are scarce in

mobile devices, and the context is continuously changing. Furthermore, the user expects his mobile applications to be smooth and fast, so the reconfiguration process needs to be very efficient and transparent to the user. Therefore, the development of reconfigurable mobile applications requires to address several important challenges:

1. **Making dynamic variation points available at runtime**. The first task of a DSPL approach is to appropriately model the dynamic variation points, that is, the elements that could be adapted dynamically. But, these dynamic variation points must be available at runtime, in order to generate the different variants of the DSPL. So, once the variation points have been specified in a variability language, such as Common Variability Language (CVL) [Object Management Group, Inc., 2012] or Feature Models (FMs) [Kang et al., 1990], the challenge is how they can be made available at runtime in order to generate the successive runtime configurations. Most DSPLs approaches apply model driven development technologies, which apply model transformation techniques to generate runtime models or code from a variability model (for instance, an FM). We propose an architecture-centric approach in which the dynamic variation points, instead of being defined in terms of the application features (as usual in FMs), are defined in terms of the elements of the application architecture. So, at runtime, we generate the successive software architecture configurations by binding the architectural variation points specified in a variability language [Pascual et al., 2013a]. Regarding the variability language used, in this thesis we propose two alternatives: the use of CVL and the use of UML profiles. CVL eases the generation of architectural variation points compared with, for instance, FMs [Pascual et al., 2013a]. In the case of UML profiles, we also propose a mechanism for reasoning about the architectural variability, enabling the detection of errors in the definition of the variability. Chapter 4 shows how both approaches have been integrated in our development methodology.

2. **Ensuring the consistency of the DSPL architecture**. In DSPL architecture-centric approaches, the software architect manually defines the architectural variability and some constraints between architectural artifacts, so it is possible that he may introduce some inconsistencies. The challenge is to provide the software architect with tool support to make the automatic checking of architectural variability inconsistencies possible. In our approach, when the variability is modelled using an UML profile such as ADOM [Reinhartz-Berger and Sturm, 2014], we can detect and solve some of these inconsistencies. This allows the software architect to refine the specification of the architecture or the variability if they are not correct.

3. **Optimizing the architectural configuration**. Any DSPL approach ensures that the successive configurations that are instantiated at runtime are valid regarding the variability model. But, sometimes this is not enough, in addition, the DSPL process must also ensure that runtime configurations are also optimal in regard to some specific criteria (e.g. user preferences, quality of service, amount of resources, etc.). In our case, the primary aim of our research is to provide dynamic adaptability of applications running on mobile devices, with resource constraints. So, we need to consider not only the valid, but also the optimal architectural configurations which do not exceed the usage of the device's resources (e.g. battery, memory, etc.). To this end, we have defined two different optimization algorithms, DAGAME [Pascual et al., 2015b] and MO-DAGAME [Pascual et al., 2015a], which are able to find nearly-optimal configurations taking into account the resource usage of the valid architectural configurations. MO-DAGAME is the multiobjective version of DAGAME, being able to optimize several criteria simultaneously. Note that an exact algorithm cannot be used for this purpose because the problem to be solved has been proven to be NP-hard (non-deterministic polynomial-time hard) [White et al., 2009].

4. **Generating the reconfiguration plan at runtime**. Most DSPL approaches generate, at design time, the configurations that will be deployed at runtime [Rosenmüller et al., 2011, Rouvoy et al., 2009, Shen et al., 2011, White et al., 2007]. However, the potential number of configurations normally grows exponentially with the number of dynamic variation points. In order to address this serious problem, some approaches consider only a subset of the valid configurations at runtime (e.g. the most probable ones), which are pre-loaded in the system. However, this is an important drawback, especially in our case. It would be very difficult to ensure at design time, that the list of loaded configurations includes the optimal ones according to the resources that are available at any point of the application's execution. So, it is preferable to automatically generate all the potential configurations at runtime, there by making it possible to choose the optimal one taking into account a given context change. Concretely, in our approach the different architectural configurations are generated on demand using the DAGAME or MO-DAGAME optimization algorithms, which are loaded in the mobile device. The reconfiguration plan is easily calculated as the difference between the running and the new configuration generated by the optimization algorithm.

5. **A scalable decision making process**. Those DSPL approaches that perform the analysis and derivation of reconfiguration plans at design time are usually based on the definition of a set of event-condition-action (ECA) rules [Cetina et al., 2008, Shen

et al., 2011]. An ECA rule includes the event that will trigger a reconfiguration, a condition about the system state that must be evaluated as true, and the reconfiguration plan or actions that have to be executed. The main problem with this approach is that the number of rules could become untreatable, especially if the number of potential configurations is very high. Goal-based approaches overcome this problem since they do not need to enumerate all the "context change - product configuration" pairs at design time, but at a cost of more runtime overhead. In our approach, DAGAME optimizes a *utility function* that quantifies the quality of the generated product configurations. Although our approach is independent of the chosen utility function, the notion of utility typically refers to the expected user's overall satisfaction. For instance, the criterion used to determine the utility of a component could be the **precision** and the **measuring rate** in the case of a component that provides location information or the **quality** in the case of a component for video streaming. Because of its ability to fit well with optimization problems based on variability, the concept of utility function has been applied before in other proposals, such as MUSIC [Rouvoy et al., 2009] and [Paspallis, 2009]. In our approach we can also use MO-DAGAME, that optimizes a multiobjective function. For instance, MO-DAGAME can generate configurations which are optimal regarding several criteria such as battery consumption, memory usage and usability. In [Pascual et al., 2015a] we demonstrate that DAGAME and MO-DAGAME are scalable, generating nearly-optimal configurations for large FMs fast enough to be used on mobile devices.

6. **Executing the service with scarce resources**. An important challenge of any reconfiguration service executing in a mobile environment is to reduce by as much as possible the resources (time, memory, CPU, battery) consumed by the service itself. In particular, for a reconfiguration service, the time is critical since, in order to be useful, applications must be reconfigured without the extra time employed for the reconfiguration process being noted. In this regard, in [Pascual et al., 2015b] we demonstrate that our approach is efficient enough to avoid harming the user response time or the performance of the system.

Chapter 2

Background

In this chapter we provide the background that is necessary to understand the work presented in this dissertation and the related work.

Section 2.1 describes DSPLs and different mechanisms to specify the variability model, which is the central part in DSPLs. In Section 2.2, GAs are introduced, explaining how problems are modelled and the different stages that can be identified during the execution of GAs.

2.1 Dynamic Software Product Lines (DSPLs)

A Software Product Line (SPL) 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."[1]. DSPLs move existing SPL engineering processes to runtime, ensuring that system adaptations lead the system to a valid state. Therefore, while in SPLs the engineering processes generate several systems of the same family at design time, a DSPL is a single system which is able to adapt its behaviour at runtime.

Variability modelling, which consists in specifying the commonalities and variabilities, is the central activity of both SPLs and DSPLs. The engineering processes of SPLs generate products by selecting specific values for the variable characteristics specified in the variability model. Therefore, the SPL engineer binds the variation points at design time considering the requirements of the intended product. In contrast, in DSPLs the variability model describes the potential range of variations that can be produced at runtime for a single product, i.e. *the dynamic variation points*, which must refer to the system architectural components.

[1] http://www.sei.cmu.edu/productlines/

Therefore, in DSPLs the system architecture supports all possible adaptations defined by the set of dynamic variation points [Hallsteinsen et al., 2008].

Then, as part of a DSPL definition the engineer must define:

1. The range of potential adaptations supported by the system in terms of architectural components.
2. An explicit representation of the valid configuration space of the system.
3. The context changes that may trigger an adaptation, i.e. the criteria to initiate a reconfiguration or decision making process.
4. The set of possible reactions to context changes that should be supported the system.

However, the way these aspects are implemented may differ greatly, as will be shown in Section 3.1.

As for the majority of DSPLs the decision to initiate a reconfiguration is made autonomously by the system (not by a human), they are considered a good technology for developing self-adapting systems such as mobile applications. In this context, most DSPL approaches share some common properties with the AC paradigm such as the monitoring of the environment and the generation of successive configurations.

Feature Models

As we already stated in the last subsection, the variability model is the central artifact of both SPLs and DSPLs. Feature models are widely used for modelling variability in SPLs. Although they are typically used in the requirements specification phase, they can be successfully applied to manage variability in other phases of the software development life cycle [Acher et al., 2011, Perrouin et al., 2012]. FMs are organised in a hierarchical structure (Figure 2.1), where each feature is decomposed into children features, which can be connected to their parent individually using optional/mandatory connectors (if the child feature is optional/mandatory) or in groups (an OR group if one or more child features can be selected or an XOR group if only exactly one child feature can be selected). Selecting a feature means that its parent should be selected too.

Figure 2.1 shows the FM of a game for mobile devices.The root feature, `MobileGame`, is decomposed in the `Sound`, `Connectivity`, `GraphicsQuality`, `GlobalScoreboard` and `Multiplayer` features. While the `GraphicsQuality` feature is mandatory and thus has to be included in all the generated configurations, the rest of them are optional – i.e. they are variation points. The `Network` and `Bluetooth` features are part of an OR group, meaning

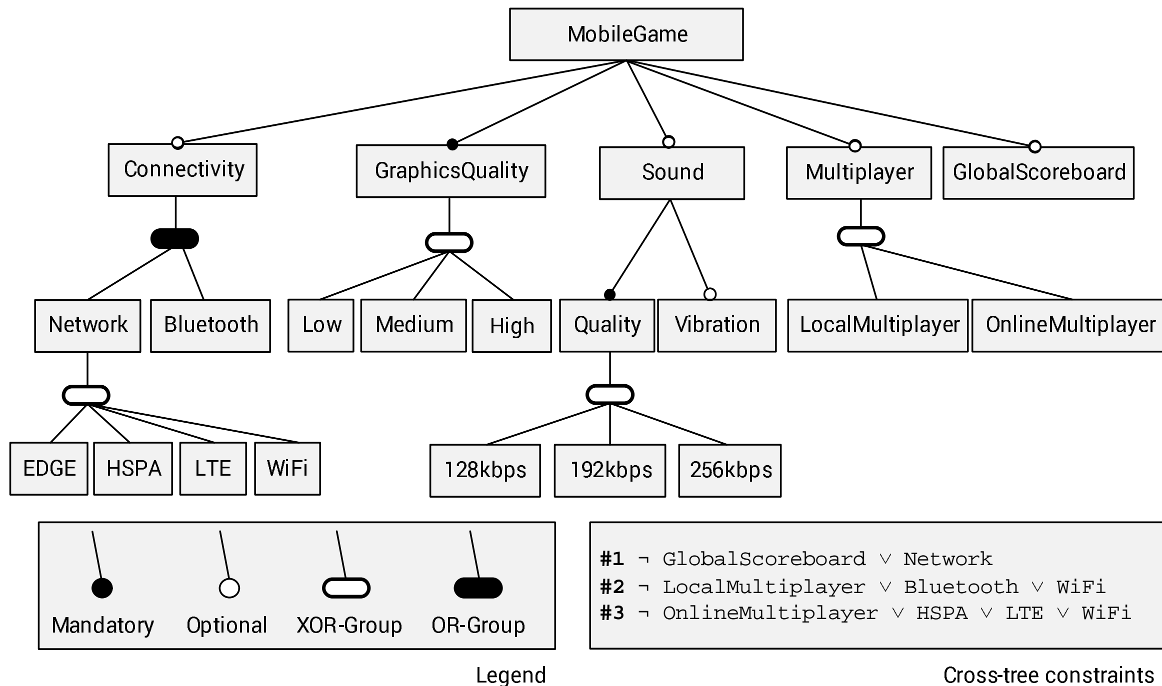


Fig. 2.1 Feature Model Example

that one or both of them can be selected simultaneously, while the `LocalMultiplayer` and the `OnlineMultiplayer` features are in an XOR group, and thus only exactly one of them can be part of a particular configuration.

In addition to the relationships between features shown in the tree (i.e. the tree constraints), it is also possible to specify Cross-Tree Constraints (CTCs) between features. In some cases, these CTCs are specified as `A requires B` or `A excludes B` statements. The first one states that, in the case that feature `A` is selected in a particular configuration of the FM, feature `B` should also be included. The second one states that the features `A` and `B` are mutually exclusive and, therefore, they cannot be selected simultaneously in the same FM configuration.

CTCs can also be defined in Conjunctive-Normal-Form (CNF) notation, which allows to define more complex constraints. In CNF, CTCs are expressed as a conjunction of clauses, where a clause is a disjunction of positive and negative literals (features); otherwise expressed, the set of CTCs is specified as a logical *AND* of *OR*s. For instance, the CTC #3 in Figure 2.1 states that, in the case that the `OnlineMultiplayer` feature is selected, it is necessary to select the `HSPA`, `LTE` or `WiFi` features:

$$OnlineMultiplayer \implies HSPA \vee LTE \vee WiFi$$

Thanks to the extensive use of FMs, it is possible to take advantage of their wide sup-

port ([Acher et al., 2010, Benavides et al., 2010, Matinlassi, 2004, White et al., 2009]) and the existing tools (e.g. FAMA [ISA Group, 2010], Hydra [CAOSD Group, 2009], SPLOT [Computer Systems Group, 2014] or FeatureIDE [Kastner et al., 2009]). Moreover, FMs are specified using formal languages, as for instance CSP (Constraint Satisfaction Problems) [Tsang, 1993]. This means that the visual representations of FMs are only for the purpose of facilitating the writing and understanding of the FMs, but then the existing tools automatically map this graphical representation into a CSP specification. This allows reasoning about variability, as well as other capacities of FMs such as the generation of valid product configurations, the quantification of the number of possible configurations, etc. [Benavides et al., 2010].

Common Variability Language (CVL)

CVL [Object Management Group, Inc., 2012] is a domain-independent language for both specifying and resolving variability. Its prime advantage is that it allows the specification of variability over any model which has been defined using a Meta-Object Facility (MOF) [Object Management Group, Inc., 2002] based metamodel. An overview of the approach proposed by CVL can be seen in Figure 2.2. By one side, the software architect specifies the *base model* of the application, which does not contain any information about variability. On the other side, the variability information is separately specified in a *variability model*, according to the CVL metamodel. In order to generate the configuration of a specific product, the SPL engineer selects a set of options in the variability model. This set of options makes it possible to bind the variation points to concrete values, and this is what is called a *resolution model* of the variability in CVL. CVL is *executable*, meaning that it is possible to automatically generate *resolved models*, which are full product models (i.e. without variability). The advantage of CVL is that these resolved models are fully specified in the base language, making it possible for them to be processed with regular base language tools. So, it is easier to adopt than other SPL approaches, since the software architect does not have to change either the architectural language, or the design tool that is normally used.

In CVL, a variability model consists of three main parts:

1. **Variation points**. Define the points of the base model that are variable and can be modified during the execution of CVL. For instance, some of the variation points supported by CVL are the *existence* of elements of the base model or the links between them, or the *value assignment* of an attribute.

2. **Variability Specification Tree (VSpec tree)**. Tree structures the elements of which (i.e. VSpec) are similar to features in feature modeling, representing choices bound to

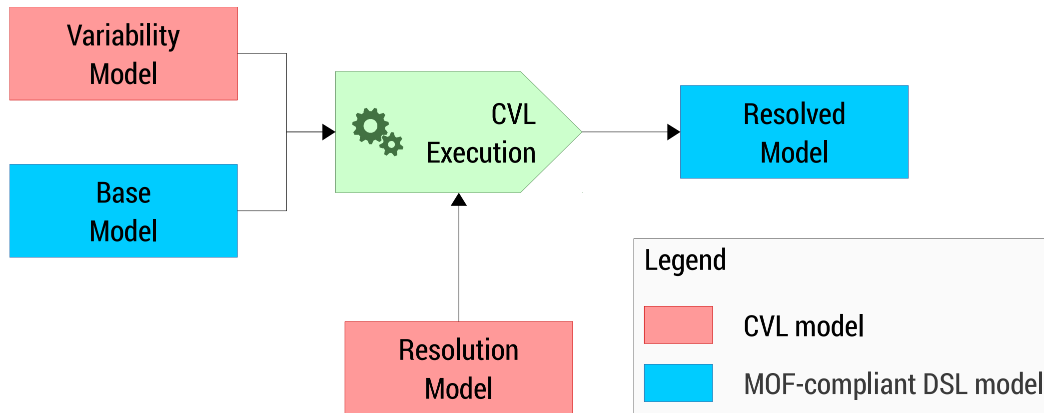


Fig. 2.2 CVL Approach

variation points. There are four types of VSpec: (1) *Choice* requires a binary decision (i.e. yes/no); (2) *variable* allows providing a value of a certain type; (3) *VClassifier* is a VSpec that can be instantiated zero or more times, and that generates a subtree for each instance. Each VClassifier has an instance multiplicity which indicates how many instances of it may be created (it is similar to the cardinality-based feature models); and (4) *composite VSpecs* (CVSpec) are used for modularity purposes. They are VSpecs that encapsulate other VSpec trees. A VSpec specification is resolved by a resolution model and propagated to the variation points and to the base model, generating the resolved model without variability.

3. **OCL Constraints**. CVL supports the definition of OCL constraints between elements of a *VSpec tree*, providing a highly flexible mechanism for delimiting the bounds of variability. These constraints are primarily used to discard invalid configurations.

2.2 Genetic Algorithms

GAs are a search heuristic, inspired by the process of evolution, which are typically used to find solutions for optimization problems. Using GAs it is possible to find nearly-optimal solutions for optimization problems without having to explore the whole solutions space. As stated by Guo et al. [Guo et al., 2011], applying GAs can be highly appropriate when the solutions space is very wide and it is not affordable to evaluate all of them due to a lack of resources and time.

In GAs, candidates to be returned as the solution to the optimization problem are known as *chromosomes*, making up a *population*. Chromosomes are typically modelled as a list of binary variables known as *genes*, each one modelling a property of the solution, although

different encodings can be defined. The function to optimize can be either mono-objective or multi-objective. In the first case, the objective function is typically used as a *fitness function* for measuring the quality of each solution, and the GA returns as the solution to the optimization problem the fittest individual in the population. In the latter case, the GA returns as a result a *front* of non-dominated solutions. A solution is non-dominated if none of the values of the objective functions can be improved without degrading the value of other objective functions.

Typically, three different stages can be identified during the execution of genetic algorithms:

1. **Generation of the initial population**. An initial set of solutions are generated to fill the population. The size of the population is a configurable parameter, and choosing the most appropriate size depends on the optimization problem that we are trying to solve.

2. **Evolution through generations**. The initial population generated in the previous step is evolved in order to find better solutions. Typically, in each generation, two chromosomes are chosen, and a *crossover* operation is performed between them, obtaining a new solution that takes genes from both chromosomes. Then, a *mutation* may be introduced in the resulting chromosome changing the value of one or more of its genes. Mutations are useful for improving the diversity of the population, but a very high mutation probability can take the optimization problem closer to a random-based search, which leads to a loss in population fitness. At the end of each generation, the new chromosome replaces the worst one in the population, thereby improving the overall fitness of the population.

3. **Returning the solution or front of solutions**. After the last generation, the best solution or a front of nondominated solutions is returned, depending on the type of optimization algorithm. Different criteria can be defined for stopping the evolution process. For instance, a maximum number of generations can be specified as a configurable parameter, or a maximum number of evaluations of the objective function. Moreover, it is also possible to stop the evolution if the algorithm is unable to improve the population fitness during a concrete number of consecutive generations.

As part of our approach we have defined two different optimization algorithms that are used to generate quasi-optimal application configurations at runtime that do not exceed the available resources: DAGAME (a mono-objective GA) and MO-DAGAME (a multi-objective GA).

Chapter 3

Related Work

This section enumerates and describes work related to self-adaptation and optimization algorithms, on which this thesis are focused. First, Subsection 3.1 describes related self-adaptation work, explaining how they address the challenges for DSPLs explained in Section 2.1. Then, Subsection 3.2 enumerates and describes different optimization algorithms used by DSPL approaches which model variability using FMs.

3.1 Self-adaptation approaches

In Chapter 1 we discussed some challenges that DSPL approaches should address and that allow us to identify the main differences betweem them. In the rest of this section we discuss how these challenges are addressed by the approaches compared here. The results of this discussion are summarized in Table 3.1, where there is a column for each challenge. In those approaches where it has been not possible to determine whether they are suitable for mobile devices, the value N/A (i.e. not available) is shown.

Firstly, we can distinguish between those approaches that generate the valid configurations at design time [Brataas et al., 2011, Gamez et al., 2011, Rouvoy et al., 2009, Shen et al., 2011, Vassev et al., 2012] and those that generate them at runtime [Ayora et al., 2012, Blouin et al., 2011, Cetina et al., 2008, Cheng et al., 2009, Gomaa and Hashimoto, 2011, Rosenmüller et al., 2011, Trinidad et al., 2007]. We can see that, among the first ones, there are important differences in how they address the challenges under discussion.

Table 3.1 Self-adaptation approaches

Approach	Config. generation	Dynamic var. points modeling	Decision making	Optimization mechanism	Mobile devices
Gamez [Gamez et al., 2011]	Design time	Feature mapping	ECA rules	None	Yes
Vassev [Vassev et al., 2012]	Design time	Custom model	ECA rules	None	N/A
Shen [Shen et al., 2011]	Design time	Custom model	ECA rules	None	No
Rouvoy [Rouvoy et al., 2009]	Design time	Custom model	Utility function	Brute force/heuristics	Yes
Brataas [Brataas et al., 2011]	Design time	Custom model	SP model	Brute force	N/A
Rosenmuller [Rosenmüller et al., 2011]	Partially runtime	Feature-oriented programming	Manual / ECA rules	None	No
Trinidad [Trinidad et al., 2007]	Runtime	Feature mapping	Manual	None	No
Ayora [Ayora et al., 2012]	Runtime	CVL	ECA rules	None	No
Cetina [Cetina et al., 2008]	Runtime	Custom model	ECA rules	None	N/A
Cheng [Cheng et al., 2009]	Runtime	Strategies	Predefined strategies	None	N/A
Gomaa [Gomaa and Hashimoto, 2011]	Runtime	Feature mapping	Utility function	Heuristics	No
Blouin [Blouin et al., 2011]	Runtime	Context-actions mapping (specific for user interfaces)	Utility function	Genetic Algorithm	N/A
Ali [Ali and Solis, 2015]	Runtime	Services interface & contracts	Utility Function	Particle Swarm Optimization	Yes
Cosmapek [Casquina et al., 2016]	Runtime	Feature mapping	SAT solver	None	Yes
Our approach	**Runtime**	**CVL**	**Utility function**	**Genetic Algorithm**	**Yes**

In particular, Gamez et al. [Gamez et al., 2011] propose a reconfiguration mechanism, also driven by the MAPE-K loop, that switches between different architectural configurations at runtime. The valid configurations are manually specified and represented using FMs, and a mapping is defined between the features in the FM and the system architecture. The reconfiguration plans are automatically generated from the differences between the configurations. Therefore, both are specified at design-time, which leads to the deployment of sub-optimal configurations at run-time. Their approach has been implemented in several languages, including Java for Android. Therefore, it can be used to develop adaptive mobile applications.

Vassev et al. propose ASCENS [Vassev et al., 2012], a framework for the representation and reasoning of knowledge, which is defined as an specific interpretation of the context data. In this framework, which enables awareness and self-adaptation, knowledge is specified using KnowLang, a language based on ontologies and Bayesian networks. The description of the system, as well as the reconfiguration polices, are specified in a single model, known as *Knowledge Representation and Reasoning (KR&R) model*. The decision making process is led by a set of predefined policies based on the execution context. Although they propose a case study which involves robots and sensors, as far as we know, their approach has not been evaluated and, therefore, it is not possible to assess its suitability in the case of mobile applications.

Shen et al. propose a dynamic reconfiguration approach [Shen et al., 2011] based on dynamic aspect weaving where the set of valid configurations is also generated at design time and the reconfiguration process is triggered by ECA rules. The variability is specified using FMs, and they propose a meta-model for specifying *role models*, which are used to bind features to elements of the software architecture. However, their approach relies on the *JBoss-AOP* framework, which is not available in mobile devices. The reason for this limitation is that *JBoss-AOP* relies on *cglib*, a Java library for runtime bytecode generation which is not available in Java virtual machines for mobile devices such as Dalvik, the virtual machine used in the Android operating system.

MUSIC [Rouvoy et al., 2009] is an OSGi-based middleware for developing context-aware adaptive applications. It is a component based and service oriented approach which principally consists of two different parts: the context and the adaptation middlewares. The adaptation middleware is responsible for adapting the applications,deploying the configuration that best fits the current context by evaluating an utility function specified by the software architect. In order to take advantage of MUSIC, the architecture and the variability of the applications are specified together in the same model, according to the meta-model proposed. The main difference between MUSIC (as well as other existing approaches) and

our approach is that they require having available at runtime all the valid configurations of an application, while in our approach this configuration is generated on demand using the optimization algorithm. Although the MUSIC middleware does not focus on mobile devices in particular, it is possible to execute it on mobile devices supporting an implementation of the OSGi platform (e.g. Android devices).

Brataas et al. [Brataas et al., 2011] propose a mechanism for extending MUSIC with support for specifying the requirements and the utility of the components of a software architecture. They estimate how many hardware operations are generated by each user action and the application response time. To this end, a structure and performance (SP) model is defined, which allows them to evaluate, at runtime, the resource usage of each configuration and therefore decide whether a configuration is appropriate for the current context. We have identified several drawbacks to this approach:

1. The authors state that it is necessary to evaluate each variant of the application at runtime, which can lead to scalability issues.

2. As far as we know, their approach has not been evaluated on mobile devices and, therefore, it is not possible to determine its suitability for reconfiguring mobile applications.

3. Some profiling tools that are necessary to obtain the resources' usage information are not available on mobile devices.

Summarizing, we can see that, on the one hand, in the majority of these approaches, the decision making process is triggered by ECA rules, and therefore they do not provide any optimization mechanism. As stated before, this leads to sub-optimal configurations at run-time. On the other hand, MUSIC is the only one of these approaches that is suitable and available for mobile devices.

Regarding those proposals which generate valid configurations at runtime, we can also find notable differences. For instance, the majority of them do not provide an optimization mechanism [Ayora et al., 2012, Cetina et al., 2008, Cheng et al., 2009, Rosenmüller et al., 2011, Trinidad et al., 2007]. Only those whose decision making process is lead by utility function [Blouin et al., 2011, Gomaa and Hashimoto, 2011] use an optimization mechanism (either heuristics or a genetic algorithm).

Rosenmuller et al. [Rosenmüller et al., 2011] present a DSPL approach which partially generates the configurations at runtime. Firstly, part of the variability of the SPL is reduced at design time, generating several DSPLs which are subsets of the complete SPL. So, an adaptation mechanism is included in each DSPL that is capable of generating different

configurations of that DSPL at runtime. A SAT solver is used to reconfigure the application at runtime and, since it is demonstrated that SAT problems are NP-complete, it is not appropriate for mobile devices.

Trinidad et al. [Trinidad et al., 2007] propose a DSPL approach where each feature in the FM is mapped to a component in the software architecture that can be activated or deactivated. Therefore, using a CSP solver, they perform a real-time analysis of the FM, generating valid configurations at runtime. In this approach, the user manually proposes new configurations of the application, and the CSP solver is used to reason about their validity, among other operations.

Ayora et al. [Ayora et al., 2012] propose a mechanism for managing variability in business processes. At design time, variability is modeled using CVL. Then, following the MAPE-K loop, process variants are adapted using *MoRE-BP*, a reconfiguration engine for web services. Instead of providing an optimization mechanism, the reconfiguration process is triggered by ECA rules. Furthermore, their approach solely focus on web services, making it unsuitable for the development of mobile applications.

In [Cetina et al., 2008] Cetina et al. present an approach for the design of pervasive SPLs which are reconfigured according to changes in the environment. The pervasive system is modelled using the Pervasive Modeling Language (PervML), which includes the ECA rules that trigger the reconfiguration process. The variability of the system is specified using FMs, and both models (PervML and FMs) are related using a realization model, which is an extension that they have incorporated in Atlas Model Weaving (AMW) [Didonet et al., 2006], an Eclipse plugin for model weaving. In order to cope with the complexity of the variability, the SPL is pruned at design time, removing those model elements which are related to scenarios that are supposed to be not useful and therefore limit the configuration space. They have evaluated their approach by applying it to a smart home case study. However, as far as we know, they do not provide details on the execution time and the scalability of their approach. Therefore, we can not determine whether it is appropriate for mobile devices.

Cheng et al. propose a predictive, instead of reactive, adaptation approach [Cheng et al., 2009], trying to foresee changes in the availability of resources and lower the disruption to the quality of service provided to the user. To this end, they extend the Rainbow framework [Cheng et al., 2005] with a mechanism for predicting resource availability based on data gathered in the past. The adaptation process is based on predefined strategies, which specify the changes that need to be applied to the system under certain conditions and contexts in particular. Furthermore, this approach has been evaluated by applying it to a web service and, therefore, it is not possible to assess the suitability of their approach in the case of mobile applications.

In [Gomaa and Hashimoto, 2011], Gomaa et al. propose a DSPL approach which enables the dynamic adaptation of software architectures, but it is exclusively focused on service-oriented architectures. The variability is modelled using FMs, the features of which are mapped to the artefacts of the software architecture. However, it does not provide details on how a configuration of the FM is selected at runtime, although it states that this process is usually human assisted. Moreover, this approach has been built using the Self-Architecting Software Systems framework (SASSY), a model-driven framework for service-oriented software systems which is implemented on top of Eclipse Swordfish. Therefore, this work is not suitable for mobile applications.

In [Blouin et al., 2011], an optimization algorithm is also used to improve user interface adaptation at runtime. An important difference is that their work is specific to a user interface architectural model, while our approach is more general because it can be applied to the architectural model of any kind of applications. In their approach, the dynamic variation points are modeled by performing a mapping between the context and the different reconfiguration actions that can be executed at runtime. They use a different optimization algorithm (NSGA-II) although, as in our case, their approach does not depend on a particular optimization algorithm and is designed to work with other algorithms. Finally, the average adaptation time of our approach is considerably lower than the reported in [Blouin et al., 2011].

Cosmapek [Casquina et al., 2016] is an approach for dynamically adapting service-oriented mobile applications, with the objective of adapting them according to changes in the availability of the services used by the application's components. Cosmapek is based on DSPLs and the MAPE-K loop, modelling the variability of the applications is using FMs containing both static and dynamic features, which are those that can be selected or deselected at runtime. The features in this FM are mapped to architectural elements (i.e., components and connectors) in the application's architectural model. The availability of the services consumed by the application is provided through *sensors*, and a change in the availability of one of these services may trigger a reconfiguration to deploy a new valid application configuration. The execution of the reconfiguration plan is performed by means of *effectors*, which use the Java Reflection API to add or remove components and connectors of the application. The planner generates valid application configurations using a SAT solver. However, unlike the approach in this thesis, its objective is not generating optimal configurations with respect to different criteria, but a valid configuration that does not use a non-available service. This approach has been evaluated by implementing an Android application which uses several services and simulating changes to their availability.

The most similar approach to ours is the work presented in [Ali and Solis, 2015]. Ap-

plications are modelled using a service-oriented architecture. Applications react to changes in their environment, and their software architecture may be reconfigured when they enter or exit an environment. Valid architectural configurations are generated at runtime based on the services interfaces as well as the contracts defined at design time between services providers and consumers. Unlike our approach, architectural configurations are generated using a swarm particle algorithm, which finds nearly-optimal configurations by optimizing an utility function. Therefore, unlike our proposal, it can not generate architectural configuration according to multiple objectives. Finally, although they focus on services for mobile devices, the scenario used to perform the evaluation is too small to assess the scalability as well as the quality of the generated configurations.

Summarizing, we have also identified significant differences among those approaches that generate the configurations at runtime. As far as we know, none of them is available for mobile devices, either because they are focused on different kinds of systems or because they use tools which are not available or suitable for these devices.

3.2 Optimization algorithms for FMs

As shown in [Pascual et al., 2015b], fast algorithms to calculate the optimum configuration at runtime are desirable. This can be modelled as an optimization problem which has proven to be NP-hard (non-deterministic polynomial-time hard) [White et al., 2008] and, therefore, exact algorithms can not be used to this end. Instead, heuristics such as algorithms based on the principles of evolutionary computation, such as genetic algorithms (GAs) or particle swarm optimization may be used.

Those DSPL approaches which model the variability using FMs need algorithms to generate configurations from FMs according to a given criterion. These algorithms mainly differ in:

1. **Efficiency for mobile device execution**. In order to be suitable for reconfiguring mobile applications at runtime, the optimization algorithm should be very efficient regarding its execution time. Furthermore, its implementation should be executable on a mobile device.

2. **Number of objectives**. We distinguish whether the algorithm can optimize one single objective or multiple objectives simultaneously.

3. **Cross-Tree Constraints support**. We evaluate the support provided by the algorithm to specify CTCs. We distinguish the cases in which no support is provided, a limited

Table 3.2 Optimization algorithms for FMs

Approach	Objectives	Mobile	CTCs
Li [Li et al., 2012]	Single	N/A	Limited
Benavides [Benavides et al., 2005]	Single	No	No
Djebii [Djebbi et al., 2007]	Single	No	Limited
Soltani [Soltani et al., 2012]	Single	No	Yes
White [White et al., 2009]	Single	Yes	Yes
Shi [Shi et al., 2010]	Single	Yes	Limited
Guo [Guo et al., 2011]	Single	Yes	Limited
Sayyad [Sayyad et al., 2013]	Multiple	No	Yes
SATIBEA [Henard et al., 2015]	Multiple	No	Yes
SMTIBEA [Guo et al., 2017]	Multiple	No	Yes
Our approach	**Single / Multiple**	**Yes**	**Yes**

support is provided (only for `A requires B` or `A excludes B` constraints), and all CTCs are supported.

Table 3.2 summarizes some approaches that use optimization algorithms, as well as other algorithms available in the literature for generating configurations of FMs.

Li et al. [Li et al., 2012] present an algorithm for transforming the problem of selecting a configuration of a FM into a 0-1 Programming problem, which can be solved using different algorithms. Although it does not support optimization based on an utility function, it can find configurations which do not exceed a certain amount of resources. With regards to CTCs, only basic CTCs are supported. Moreover, since an evaluation of their approach is not provided, it is not possible to assess whether it is suitable for mobile devices or not.

In the work of Benavides et al. [Benavides et al., 2005], FMs are specified as a CSP problem. Then, a solver is used to analyze the variability of the FM, as well as to find optimal configurations regarding a given criteria. However, we can identify two important drawbacks to this approach: (1) it does not support CTCs, and (2) it optimizes a problem in which the complexity increases exponentially with respect to the number of features, using a CSP solver. Therefore, it is not suitable for runtime reconfiguration because the CSP solver generates an exact solution.

The work of Djebii et al. [Djebbi et al., 2007] provides support for finding optimal configurations of an FM regarding a given criteria, such as the cost of implementation, allowing in addition to discard those configurations which do no satisfy a given set of re-

quirements. However, only basic CTCs are supported and the solver is implemented in GNU-Prolog [Diaz and Codognet, 2000], which is not available for mobile devices.

Soltani et al. [Soltani et al., 2012] propose a method for finding configurations of FMs taking into account functional and non-functional requirements. To this end, Hierarchy Task Network Planning is used [Sacerdoti, 1975]. Although it provides full support for CTCs, it is not suitable for mobile devices since the reconfiguration time is very high, even when it is executed on a desktop computer.

In the work of White et al. [White et al., 2009] Filtered Cartesian Flattening [White et al., 2008] is applied to find configurations of an FM which are optimal regarding a single objective. It is mentioned in their paper that CTCs are supported, but it remains unclear which kind of CTCs are supported as the case study provided does not contain any CTC. Their evaluation results show that their approach provides nearly-optimal configurations and the execution time of the algorithm is very low. Even having a good execution time, as just said the algorithm is not multiobjective. The work of Shi et al. [Shi et al., 2010] is a slight variation of the approach presented by White et al. [White et al., 2009]. Although it is stated that their proposal is faster, the conclusions appear to be contradictory and an evaluation comparing both algorithms is not provided.

Guo et al. [Guo et al., 2011] specify a genetic algorithm to find nearly-optimal configurations of an FM taking into account a single objective. In this work, valid configurations are generated using a `fix` operator, but only basic CTCs are supported. A comparison with the work of White et al. [White et al., 2009] is provided, and the results show that the execution time of the algorithm of Guo et al. is lower, but at the cost of generating slightly worse configurations. Once again, it is not valid for our purposes because it is a single objective algorithm.

Sayyad et al. [Sayyad et al., 2013] use several MOEAs that are extensively used to find configurations of FMs, which can be optimized regarding different criterion simultaneously. In this work, all CTCs are supported, and a `fix` operator is also used, but for a different purpose. In our approach, as well as in the approach of Guo et al. [Guo et al., 2011], the goal of our `fix` operator is to repair an infeasible configuration, generating a valid one. However, the goal of the `feature fixing` operator of Sayyad et al. is to ensure that a list of features, which are common to all the valid configurations, are going to be present in all the possible configurations generated by the MOEA. As a result, invalid configurations are also generated, which are not useful for our purpose. Finally, as is shown in [Pascual et al., 2015a], in the work of Sayyad et al., as well as in our approach, a seeding technique is applied to generate the initial population.

SATIBEA [Henard et al., 2015] combines multi-objective search-based optimization

with a SAT solver that adds support for evaluating CNF constraints. It has been evaluated using several large real-world FMs, demonstrating that the quality indicators of the results outperform related approaches. However, it is not suitable for mobile devices because, in the case of large FMs, it takes several minutes in a desktop computer to reach stable quality indicators in the Pareto front.

SMTIBEA [Guo et al., 2017] is another hybrid multi-objective optimization algorithm that, unlike SATIBEA, combines Satisfiability Module Theories (SMT) and IBEA. Concretely, SMT is introduced as part of the IBEA's mutation operator and of the generation of the initial population, with the objective of supporting a richer constraints expressiveness (i.e., non-Boolean constraints) than SATIBEA. However, SMTIBEA is also not suitable for adapting mobile applications at runtime because, as shown in its experimental evaluation, using an SMT solver is expensive in terms of execution time.

Chapter 4

Approach Overview

We have defined an approach that, on the one hand, enables the modelling of reconfigurable applications (design time) and, on the other hand, provides a dynamic reconfiguration service to reconfigure applications at runtime according to the current context (runtime).

An overview of our work, which addresses all the challenges shown in Chapter 1, can be seen in Figure 4.1. In the design time phase, we can distinguish 3 different stages in the process of modelling the reconfigurable application. In the first stage (step D1 in Figure 4.1), the software architecture is specified, including all the architectural artefacts (e.g. components and connectors).

Then, in the second stage (step D2) the variability of the software architecture is specified choosing between two different approaches, addressing *Challenge 1*. The first alternative explored during the development of this thesis is the flow labeled with D2.1 in Figure 4.1. We can use a UML profile, such as ADOM, to model the variability. In this case, the variability is specified together with the software architecture using stereotypes, with the advantage that UML is well known by SPL practitioners. However, the main drawback is that the tool support that is necessary to manage the architectural variability using UML profiles is not as mature as the support already provided by existing FM tools. Therefore, we provide a mapping algorithm to convert the software architecture with variability to an FM where the features in the tree represent architectural artefacts, an architectural feature model (AFM). The software architect can then take advantage of existing FM tools to analyse the variability of the software architecture without needing to manipulate the AFM at all. Instead, he has only to interpret the results of the FM tools, which will be provided in terms of architectural artefacts. This allows the software architect to detect inconsistencies in the specification of the variability and to reason about the variability degree, addressing *Challenge 2*. In the case that some inconsistencies are detected, the variability specification can be refined and analysed again, until no inconsistencies are detected.

Alternatively, variability can also be specified using CVL (label D2.2 in Figure 4.1). In this case, variability is modelled separately from the software architecture, which also has some advantages. Firstly, the variability can be specified over any model which has been defined using a MOF-based metamodel. Secondly, the application variant obtained once the variability has been resolved is specified using the same language than was used to specify the software architecture. Therefore, the software architect does not need to use a different architectural language or a different design tool.

Although both alternatives are appropriate for specifying the DSPL variability, each one has its own advantages and choosing one of them depends on the particular requirements or constraints of the application specification. For instance, in the case that the software architecture is not specified in UML or a UML profile for variability specification can not be used, the alternative D2.2 can be followed instead. This alternative does not only not require modifying the software architecture to introduce variability specifications, but it also allows modelling the software architecture using different metamodels. On the other hand, if UML profiles for variability specification can be used, the alternative D2.1 enables the iterative refinement of the software architecure thanks to the use of FM tools.

Once the architectural variability has been specified, the third stage in our approach (label D3) is to specify the data that will be used to optimize the applications' configurations at runtime (addressing Challenge 3). To this end, we provide two different choices:

1. Defining an utility function, together with information about the resource usage of the different elements of the software architecture (e.g. battery consumption or memory usage). This function will be used at runtime by the DAGAME algorithm to generate configurations that optimize the utility while not exceeding the available resources.

2. Defining a multiobjective function, with several objectives such as usability, performance, memory usage, network usage, etc. In this case, the MO-DAGAME algorithm will be used at runtime to generate a front of configurations that optimize the values of these objective functions.

Once the software architecture, the architectural variability and the optimization data have been specified, the design time phase is completed.

At runtime, in regard to the reconfiguration of the applications, we follow the widely known MAPE-K loop [Kephart and Chess, 2003] of the Autonomic Computing [IBM, 2005] (AC) paradigm, where "M" stands for Monitor, "A" for Analyse, "P" for Plan, "E" for Execute and, finally, "K" for Knowledge. In particular, we propose a middleware in which a **Context Monitoring Service** (CMS) and a **Dynamic Reconfiguration Service** (DRS) provide support for deploying reconfigurable applications. The different stages of the MAPE-K

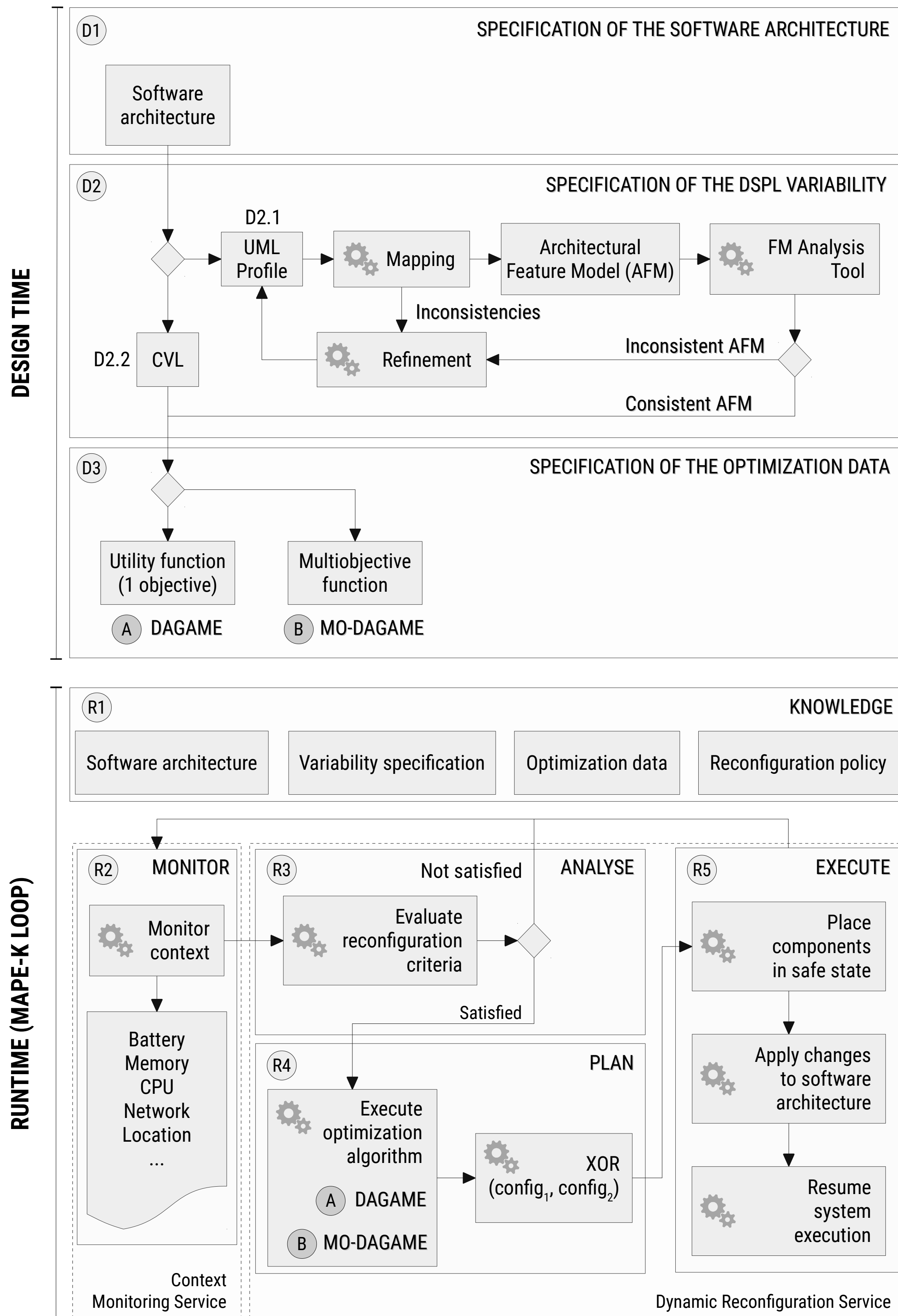


Fig. 4.1 Approach overview

loop are addressed as follows:

1. **Knowledge (R1)**. In our approach, knowledge is represented by: (1) the software architecture; (2) the variability specification; (3) the optimization data and (4) the reconfiguration policy. The variability specification will vary depending on the mechanism chosen for specifying the variability. For instance, as is shown in [Pascual et al., 2015b], in the case of CVL, the variability specification consists of the dynamic variation points, the VSpecs tree and the OCL constraints. In the case of using an UML profile, it will consist of the AFM. The optimization data is represented by the utility function or the multiobjective function, and the reconfiguration policy specifies whether a change in the execution context should trigger or not the reconfiguration process.

2. **Monitor (R2)**. The CMS provides the DRS with information about the evolution of the availability of a certain resource, such as the battery level or the memory. When a change is detected, the DRS is notified.

3. **Analyse (R3)**. When a change is detected in the execution context, the DRS analyses whether it is significant enough to trigger the reconfiguration process –i.e. if the reconfiguration criteria is satisfied. There can be several criteria for measuring the significance of a context change. For instance, a change in the battery level can be significant if it has changed by more than 5% since the last measurement was taken, or if it changes by more than 10% per hour. Therefore, several reconfiguration policies can be defined, and the policy applied is part of the *Knowledge* base.

4. **Plan (R4)**. In the case that the analyser decides that the application needs to be adapted, the optimization algorithm (either DAGAME or MO-DAGAME) is executed to find a configuration which is appropriate for the current context (addressing *Challenge 4*). Then, the differences between the current configuration and the new one are calculated, generating a plan for switching between them. Calculating the difference between two configurations is quite straightforward since it is directly obtained by performing an XOR operation between both configurations. In both CVL and AFMs, configurations are encoded as a sequence of ones and zeros, and therefore the generation of the software architecture of the resulting configuration is very efficient, addressing *Challenge 5*.

5. **Execute (R5)**. Finally, the reconfiguration plan is executed in order to adapt the running architecture of the application, which implies removing the components that are

no longer needed, adding the new components, connecting them and also reconfiguring the modified parameters. To ensure that this process is performed flawlessly, all the components are placed in a safe state before they are reconfigured. Then, once the reconfiguration plan has been executed, the components are activated and the system execution resumes.

In order to further address *Challenge 5*, we have also focused on implementing all the functions efficiently, reducing by as much as possible both the resources consumed and the response time.

Chapter 5

Discussion of Results

Within the scope of this thesis we have defined a methodology to specify reconfigurable mobile applications and a middleware platform to reconfigure these applications at runtime, deploying optimal configurations for the current context. The rest of this section details the contributions of this work, specifying the publications were they were presented:

1. A mechanism for specifying the variability of mobile applications using a UML profile and making it available at run-time. To this end, we have defined an Architectural Feature Model, where the elements of the tree represent architectural artefacts, and implemented a mapping algorithm that converts the software architectural model to an FM which can be analysed at run-time. This contribution is presented in [Pascual et al., 2013a].

2. Definition of an iterative process based on FM tools to detect inconsistencies in the specification of the variability and to reason about the variability degree. This contribution is presented in [Pascual et al., 2013a].

3. A mechanism for using the architectural dynamic variability information as entry to a genetic algorithm, which allows us to define the DRS and the CMS. This contribution is presented in [Pascual et al., 2013b].

4. A mechanism for specifying the variability of reconfigurable mobile applications in CVL. This allows the software architect to model the software architecture of reconfigurable applications using the same language and the same architectonic tools than in the rest of applications. This contribution, together with an overview of our approach, is presented in [Pascual et al., 2015b].

5. Definition of DAGAME, a genetic algorithm which enables the generation, at runtime, of quasi-optimal configurations of the applications that do not exceed the available resources, by optimising an utility function. It has been implemented for the Android platform as part of the DRS. It is presented and evaluated in [Pascual et al., 2015b].

6. Definition of a multiobjective genetic algorithm, MO-DAGAME, presented in [Pascual et al., 2015a], that generates a front of configurations of the mobile application that optimize a set of objective functions while being efficient enough to be used on mobile devices. A study has been performed to choose the most appropriate MOEA depending on the size of the software architecture and the number of objectives to optimize.

7. Both DAGAME and MO-DAGAME have been applied to several case studies [Pascual et al., 2015a,b]. They have been evaluated, obtaining good results in both cases with respect to the criteria of execution time and the quality of the obtained solutions. With respect to the execution time, we have demonstrated that they can be executed in mobile devices with scarce resources. Firstly, we show that DAGAME generates, in less than 22 ms, valid architectural configurations whose utility is greater than 87% of the utility of the optimal configuration obtained using exact methods, which are not suitable for mobile devices. However, as the optimality and execution time of DAGAME has been evaluated by simulating the execution of a mobile application, additional experimentation would be necessary to assess the generalisability of the approach presented in this thesis:

   (a) Finding the exact solution is an NP-hard problem and is not feasible for mobile applications with a high degree of variability. However, in the case of applications with a low number of valid configurations, it would be worth comparing the quality of the solutions and the performance of DAGAME with those obtained with exhaustive search and random-based search.

   (b) Evaluating this approach on larger applications with a higher number of valid configurations of the software architecture.

   (c) Evaluating this approach in real execution environments, running mobile applications on mobile devices in environments with a changing context.

   Secondly, we demonstrate that MO-DAGAME can be used in mobile devices to find valid architectural configurations of real use cases in less than 114 ms, optimising them with respect to three different objectives (usability, battery consumption and

memory footprint). Given that there is a trade-off between the execution time and the quality of the obtained solutions (their proximity to the true Pareto front of solutions), we provide guidelines to choose the most suitable MOEA depending on the use case.

Chapter 6

Conclusions and Future Work

This thesis presents an approach for modelling and reconfiguring mobile applications, which covers both design-time and run-time. At design time, the software architecture, the DSPL variability and the optimization data are specified. Two different mechanisms are proposed for modelling the DSPL variability. On the one hand, it can be modelled together with the software architecture using a UML profile [Pascual et al., 2013a]. In this case, the software architecture with variability is automatically mapped to an Architectural Feature Model (AFM) using the algorithm presented in [Pascual et al., 2013a]. This AFM is then used as input to an FM analysis tool to detect inconsistencies in the definition of the variability and refine the specification. On the other hand, it can be modelled separately from the software architecture using CVL, which allows us to:

1. Specify the software architecture using a MOF-based metamodel.

2. Obtain the resolved application variants in the same language used to specify the software architecture [Pascual et al., 2015b].

With respect the specification of the optimization data, two different choices are also proposed:

1. Defining an utility function, together with information about the resource usage of the elements of the software architecture.

2. Defining a multiobjective function, allowing us to generate a front of configurations optimized with respect to several objectives (e.g, performance, battery usage, memory usage, etc.) simultaneously.

At runtime, a middleware is proposed that follows the MAPE-K loop of Autonomic Computing by means of a Context-Monitoring Service (CMS) and a Dynamic Reconfiguration Service (DRS). The CMS is responsible for providing the DRS with information about changes in the availability of resources such as the battery level or memory. The DRS, on the other hand, analyses the changes in the context and triggers the reconfiguration process if necessary. Using an optimization algorithm, it generates an architectural configuration tailored to the current context. A reconfiguration plan is then calculated and executed to switch to the new configuration.

With regard to the algorithm for generating architectural configurations, two different choices are provided: DAGAME [Pascual et al., 2015b] and MO-DAGAME [Pascual et al., 2015a]. DAGAME is a genetic algorithm that generates valid and nearly-optimal configurations that do no exceed the available resources by optimizing an utility function. On the other hand, MO-DAGAME is a multiobjective evolutionary algorithm that generates a front of configurations which are optimized with respect to several objectives.

This approach, including DAGAME and MO-DAGAME algorithms, has been evaluated using different case studios, demonstrating that it can be used to generate nearly-optimal application configurations and that it is efficient enough to be executed on mobile devices.

As part of future work, there are several research lines that could be explored, such as:

1. Extending our software variability specification process with support for detection of variability inconsistencies when CVL is used as the variability modelling language.

2. Evaluating additional types of algorithms for generating nearly-optimal architectural configurations at runtime.

3. Implementing and evaluating our middleware platform and optimization algorithms in new types of devices and use cases.

Part II

Publications Composing the PhD Thesis

Chapter 7

Self-adaptation of Mobile Systems Driven by the Common Variability Language

Title	Self-adaptation of Mobile Systems Driven by the Common Variability Language
Authors	Gustavo G. Pascual, Monica Pinto, Lidia Fuentes
Impact Factor	2.430 (Q1)
Journal	Future Generation Computer Systems
Publisher	Elsevier
Publication Date	June 2015
DOI	http://dx.doi.org/10.1016/j.future.2014.08.015

Abstract. The execution context in which pervasive systems or mobile computing run changes continually. Hence, applications for these systems require support for self-adaptation to the continual context changes. Most of the approaches for self-adaptive systems implement a reconfiguration service that receives as input the list of all possible configurations and the plans to switch between them. In this paper we present an alternative approach for the automatic generation of application configurations and the reconfiguration plans at runtime. With our approach, the generated configurations are optimal as regards different criteria, such as functionality or resource consumption (e.g. battery or memory). This is achieved by: (1) modelling architectural variability at design-time using the Common Variability Language (CVL), and (2) using a genetic algorithm that finds nearly-optimal configurations at run-time using the information provided by the variability model. We also specify a case study and we use it to evaluate our approach, showing that it is efficient and suitable for devices with scarce resources.

Chapter 8

Applying Multiobjective Evolutionary Algorithms to Dynamic Software Product Lines for Reconfiguring Mobile Applications

Title	Applying Multiobjective Evolutionary Algorithms to Dynamic Software Product Lines for Reconfiguring Mobile Applications
Authors	Gustavo G. Pascual, Roberto E. Lopez-Herrejon, Monica Pinto, Lidia Fuentes, Alexander Egyed
Impact Factor	1.424 (Q1)
Journal	Journal of Systems and Software
Publisher	Elsevier
Publication Date	May 2015
DOI	http://dx.doi.org/10.1016/j.jss.2014.12.041

Abstract. Mobile applications require dynamic reconfiguration services (DRS) to self-adapt their behavior to the context changes (e.g., scarcity of resources). Dynamic Software Product Lines (DSPL) are a well-accepted approach to manage runtime variability, by means of late binding the variation points at runtime. During the system's execution, the DRS deploys different configurations to satisfy the changing requirements according to a multiobjective criterion (e.g., insufficient battery level, requested quality of service). Search-based software engineering and, in particular, multiobjective evolutionary algorithms (MOEAs), can generate valid configurations of a DSPL at runtime. Several approaches use MOEAs to generate optimum configurations of a Software Product Line, but none of them consider DSPLs for mobile devices. In this paper, we explore the use of MOEAs to generate at runtime optimum configurations of the DSPL according to different criteria. The optimization problem is formalized in terms of a Feature Model (FM), a variability model. We evaluate six existing MOEAs by applying them to 12 different FMs, optimizing three different objectives (usability, battery consumption and memory footprint). The results are discussed according to the particular requirements of a DRS for mobile applications, showing that PAES and NSGA-II are the most suitable algorithms for mobile environments.

Chapter 9

Automatic Analysis of Software Architectures with Variability

Title	Automatic Analysis of Software Architectures with Variability
Authors	Gustavo G. Pascual, Monica Pinto, Lidia Fuentes
Impact Factor	CORE A
Conference	13th International Conference on Software Reuse (ICSR 2013)
Publisher	Spring Berlin Heidelberg
Publication Date	June 2013
ISBN	978-3-642-38977-1
DOI	http://dx.doi.org/10.1007/978-3-642-38977-1_9

Abstract. Software Product Line Engineering is successfully applied in the development of families of related products. Basically, it allows reusing the software artifacts that are common to all the products, and adding/removing the variable ones. There are two alternatives to manage variability, one that models the commonalities and variabilities separately from the software product line architecture (SPLA), using, for instance, feature models (FM), and another one that models the variability as part of the SPLA. These two alternatives have both benefits and limitations. Our approach picks the best of both alternatives and, on the one hand, models variability as part of the SPLA (as in the second alternative), but, on the other hand, maps the SPLA with variability into an FM, generating an Architectural FM. By doing this our approach takes advantage of the FM tools and formal reasoning (as in the first alternative) to provide the automatic support that it is not available in other SPLA with variability approaches to: (i) check the consistency of architectural variability specifications, (ii) generate valid architectural configurations, and (iii) reason about variability at the architectural level.

Chapter 10

Run-Time Adaptation of Mobile Applications using Genetic Algorithms

Title	Run-Time Adaptation of Mobile Applications using Genetic Algorithms
Authors	Gustavo G. Pascual, Monica Pinto, Lidia Fuentes
Conference	8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2013)
Publisher	IEEE
Publication Date	May 2013
ISBN	978-1-4799-0344-3
DOI	http://dx.doi.org/10.1109/SEAMS.2013.6595494

Abstract. Mobile applications run in environments where the context is continuously changing. Therefore, it is necessary to provide support for the run-time adaptation of these applications. This support is usually achieved by middleware platforms that offer a context-aware dynamic reconfiguration service. However, the main shortcoming of existing approaches is that both the list of possible configurations and the plans to adapt the application to a new configuration are usually specified at design-time. In this paper we present an approach that allows the automatic generation at run-time of application configurations and of reconfiguration plans. Moreover, the generated configurations are optimal regarding the provided functionality and, more importantly, without exceeding the available resources (e.g. battery). This is performed by: (1) having the information about the application variability available at runtime using feature models, and (2) using a genetic algorithm that allows generating an optimal configuration at runtime. We have specified a case study and evaluated our approach, and the results show that it is efficient enough as to be used on mobile devices without introducing an excessive overhead.

Part III

Appendices

Appendix A

Resumen en español

Los teléfonos móviles inteligentes (*smartphones*) son una herramienta indispensable en nuestra vida cotidiana. En la actualidad, son dispositivos con los que podemos ejecutar aplicaciones y tareas complejas en cualquier lugar y en cualquier momento. Sin embargo, la computación móvil no está limitada a los smartphones. Por ejemplo, dispositivos como los relojes inteligentes (e.g., Apple Watch, Google Wear OS) son cada vez más populares y, al igual que los teléfonos móviles, ejecutan aplicaciones que están fuertemente relacionadas con su contexto (e.g., localización, recursos disponibles, etc.). Todas estas aplicaciones se podrían beneficar sustancialmente de la aplicación de mecanismos de reconfiguración dinámica.

Desarrollar aplicaciones que se adaptan al contexto en el que se ejecutan es fundamental para satisfacer los requisitos del usuario, como se ilustra en el siguiente ejemplo. Supongamos que un turista está usando su smartphone mientras explora la ciudad que está visitando. Escucha su música favorita mediante un servicio de streaming, mientras otra aplicación le notifica de los puntos de interés cercanos y de los sitios interesantes a los que se aproxima (e.g., restaurantes, tiendas...). Al mismo tiempo, otra aplicación comparte automáticamente con sus amigos las fotos que está tomando mientras pasea por la ciudad. Este usuario quiere, por supuesto, que la batería de su teléfono móvil no se agote antes de volver a su habitación del hotel. Sin embargo, durante su ruta, la batería cae a un nivel bajo, en el que no tardará mucho en agotarse. A pesar de esto, sería posible prolongar la duración de la batería, al mismo tiempo que se satisfacen los requisitos de usuario, mediante la reconfiguración de las aplicaciones que ejecuta su teléfono. Por ejemplo, la aplicación de streaming de música podría reducir la calidad de sonido o reproducir únicamente música almacenada localmente; la aplicación de información turística podría usar mecanismos de localización basados en redes inalámbricas en lugar de en satélites GPS; la aplicación para compartir fotos podría reducir el tamaño y la calidad de las mismas, o compartir sólo las fotos más relevantes.

En resumen, reconfigurar las aplicaciones móviles contribuiría a satisfacer los requisitos de usuario reduciendo, al mismo tiempo, el consumo de batería.

Un enfoque ampliamente aceptado para gestionar la variabilidad de las aplicaciones en tiempo de ejecución son las Líneas de Producto Software Dinámicas (DSPLs). Las DSPLs son capaces de producir software capaz de adaptarse a los cambios, aplicando los puntos de variabilidad en tiempo de ejecución [Hallsteinsen et al., 2008].

Por otro lado, otro paradigma ampliamente aceptado en la comunidad de los sistemas distributidos es el de la Computación Autónoma (CA) [IBM, 2005], cuyo principal objetivo es dotar a los sistemas distribuidos de capacidades de auto-gestión.

Según los principios de la CA, la reconfiguración de aplicaciones en tiempo de ejecución implica: (1) **monitorizar** el entorno en el que se ejecuta; (2) **analizar** la información monitorizada; (3) generar el **plan** de reconfiguración y (4) **ejecutar** dicho plan.

Todas estas etapas están dirigidas por una base de conocimiento, formando todas ellas el bucle conocido como MAPE-K [Kephart and Chess, 2003] del paradigma de la Computación Autónoma, donde "M" se refiere a la Monitorización, "A" al Análisis, "P" a la Planificación, "E" a la Ejecución y "K" al Conocimiento (*Knowledge* en el idioma inglés). Sin embargo, la implementación de cada una de las partes de este bucle es un problema abierto.

En nuestra propuesta, nosotros definimos un **Servicio de Monitorización del Contexto** (SMC), que se encarga de monitorizar el entorno y proporcionar esta información al **Servicio de Reconfiguración Dinámica** (SRD), que cubre el análisis de la información monitorizada y la generación y ejecución de los planes de reconfiguración. Ambos servicios están diseñados para ser integrados en un middleware para desarrollo de aplicaciones con capacidad de reconfiguración automática.

A.1 Antecedentes

En esta sección se incluye la información necesaria para poder entender el trabajo que se presenta.

A.1.1 Líneas de Producto Software Dinámicas

Una Línea de Producto Software (SPL) es “un conjunto de sistemas intensivos en software que comparten un conjunto común de características que satisfacen las necesidades específicas de un segmento del mercado o misión y que son desarrolladas, de una forma dada, a

partir de un conjunto común de recursos principales.”[1].

Las DSPLs llevan los procesos de ingeniería de SPLs existentes a tiempo de ejecución, asegurándose de que las adaptaciones llevan al sistema a un estado válido. Por lo tanto, mientras que en las SPLs, los procesos de ingeniería generan varios sistemas de la misma familia en tiempo de diseño, una DSPL es un único sistema que es capaz de adaptar su comportamiento en tiempo de ejecución.

El modelado de la variabilidad, que consiste en la especificación de las partes comunes y aquellas que son variables, es la actividad principal tanto de SPLs como de DSPLs. Los procesos de ingeniería de las SPLs generan productos mediante la selección de valores específicos para las características variables especificadas en el modelo de variabilidad. En las DSPLs, el modelo de variabilidad describe el conjunto de variaciones que potencialmente podrían generarse en tiempo de ejecución para un único producto, es decir, *los puntos de variabilidad dinámicos*, que deben estar relacionados a los componentes arquitectónicos del sistema. Por lo tanto, en las DSPLs, la arquitectura del sistema soporta todas las posibles adaptaciones definidas por el conjunto de puntos de variabilidad dinámicos [Hallsteinsen et al., 2008].

Por lo tanto, como parte de la definición de la DSPL, el ingeniero debe definir:

1. el rango de posibles adaptaciones soportadas por el sistema, en términos de componentes de la arquictectura software.

2. Una representación explícita del espacio de configuraciones válidas del sistema.

3. Los cambios de contexto que pueden iniciar el proceso de adaptación.

4. El conjunto de posibles reacciones a cambios del contexto que debería ser soportado por el sistema.

Dado que en la mayoría de DSPLs, la decisión de iniciar el proceso de reconfiguración es tomada autónomamente por el sistema (y no por un humano), son consideradas como una tecnología apropiada para el desarrollo de sistemas auto-adaptativos, como las aplicaciones móviles. En este contexto, la mayoría de las propuestas basadas en DSPLs comparten ciertas propiedades con el paradigma de la computación autónoma, como la monitorización del entorno y la generación de sucesivas configuraciones.

[1] http://www.sei.cmu.edu/productlines/

Modelos de Características (FMs)

Los modelos de características son ampliamente empleados en la definición del modelo de variabilidad de las SPLs. Aunque es común usarlos en la fase de especificación de requisitos, pueden ser aplicados satisfactoriamente en la gestión de la variabilidad en otras fases del ciclo desarrollo del software [Acher et al., 2011, Perrouin et al., 2012]. Los FMs se organizan en una estructura jerárquica (ver Figura A.1), donde cada característica se descompone en características hijas, que se conectan a su característica padre mediane conectores opcional/obligatorio (dependendiendo de si la característica hija es opcional u obligatoria) o en grupos (un grupo OR si una o más características hijas pueden ser seleccionadas simultáneamente, o un grupo XOR si sólo se puede elegir exactamente una característica hija). Seleccionar una característica implica obligatoriamente seleccionar también su padre.

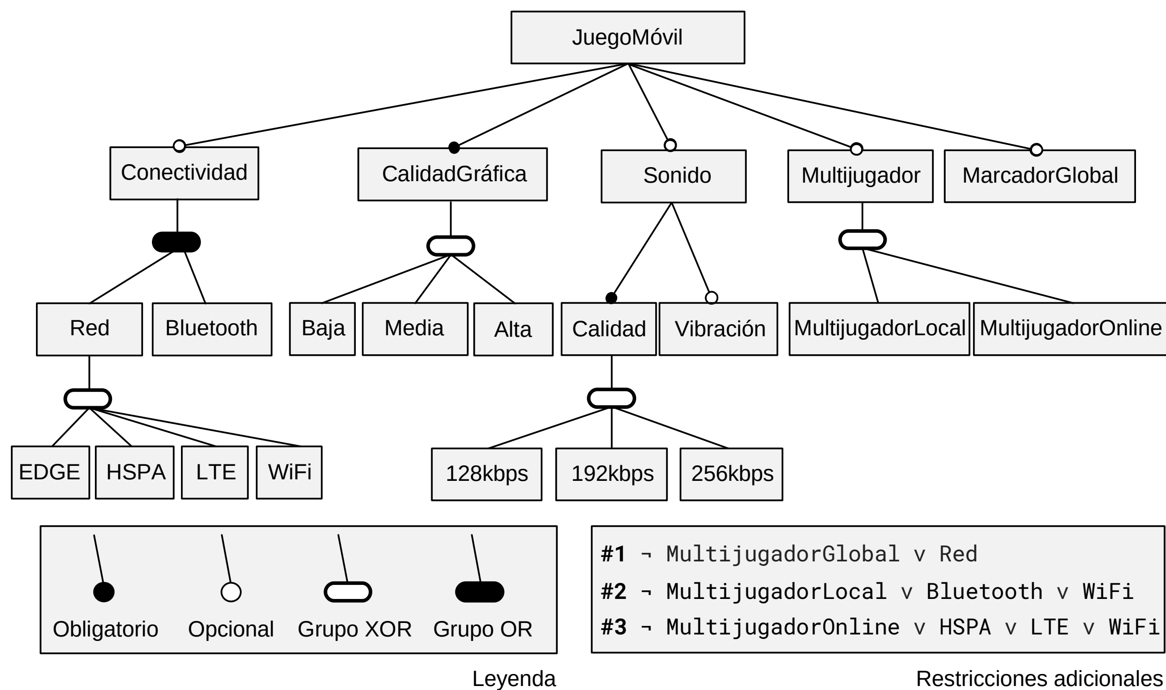


Fig. A.1 Ejemplo de Modelo de Características

La Figura A.1 muestra el FM de un juego para dispositivos móviles. La característica raíz, `JuegoMóvil`, se divide en las características `Sonido`, `Conectividad`, `CalidadGráfica`, `Multijugador` y `MarcadorGlobal`. Aunque la característica `CalidadGráfica` es obligatoria y por lo tanto debe ser incluida en todas las configuraciones generadas, el resto de ellas son opcionales (es decir, son puntos de variabilidad). Las características `Red` y `Bluetooth` forman parte de un grupo OR, lo que indica que una de ellas, o ambas, pueden ser seleccionadas simultáneamente. Sin embargo, `MultijugadorLocal` y `MultijugadorOnline`

forman parte de un grupo XOR y, por lo tanto, sólo una (y exactamente una) de ellas puede formar parte de una configuración en particular.

Además de las relaciones entre características definidas en el árbol, también es posible definir restricciones entre características no relacionadas directamente. Estas restricciones pueden ser, por ejemplo, de los tipos `A requiere B` o `A excluye B`. En el primer caso indicaría que, en el caso de que la característica `A` fuera seleccionada en una configuración del FM, la característica `B` también debería ser incluida. La segunda restricción indica que las características `A` y `B` son mutuamente exclusivas y, por lo tanto, no podrían ser seleccionadas simultáneamente en la misma configuración del FM.

Estas restricciones también pueden definirse en Forma Normal Conjuntiva (CNF), que permite definir restricciones más complejas. En CNF, las restricciones se expresan como una conjunción de cláusulas, donde una cláusula es una disjunción de literales (características) positivos y negativos. Por ejemplo, la restricción #3 en la Figura indica que, en el caso de que la característica `MultijugadorOnline` esté seleccionada, es necesario seleccionar `HSPA`, `LTE` ó `WiFi`:

$$MultijugadorOnline \implies HSPA \vee LTE \vee WiFi$$

Common Variability Language (CVL)

CVL [Object Management Group, Inc., 2012] es un lenguaje independiente del dominio para especificar y resolver la variabilidad. Su principal ventaja es que permite especificar la variabilidad sobre cualquier modelo que haya sido definido usando un metamodelo basado en *Meta-Object Facility* (MOF) [Object Management Group, Inc., 2002].

La Figura A.2 muestra una vista general del enfoque propuesto por CVL. Por un lado, el arquitecto software especifica el *modelo base* de la aplicación, que no contiene ninguna información acerca de la variabilidad. Por otro lado, la información de variabilidad se especifica de forma separada en un *modelo de variabilidad*, de acuerdo con el metamodelo de CVL. Para poder generar la configuración de un producto en particular, el ingeniero de la SPL selecciona un conjunto de opciones en el modelo de variabilidad. Este conjunto de opciones hace posible asociar valores concretos a los puntos de variabilidad, dando lugar a lo que se conoce como el *modelo de resolución* de la variabilidad en CVL. CVL es *ejecutable* y, por lo tanto, es posible generar automáticamente *modelos resueltos*, que son modelos completos del producto (es decir, sin variabilidad). Una ventaja importante de CVL es que estos modelos resueltos están completamente especificados en el lenguaje base, lo que permite procesarlos con las herramientas habituales para estos lenguajes. Por lo tanto, es más fácil adoptar CVL que otros enfoques de SPLs, ya que el arquitecto software no necesita cambiar ni el lenguaje de la arquictectura software ni la herramienta que normalmente usa.

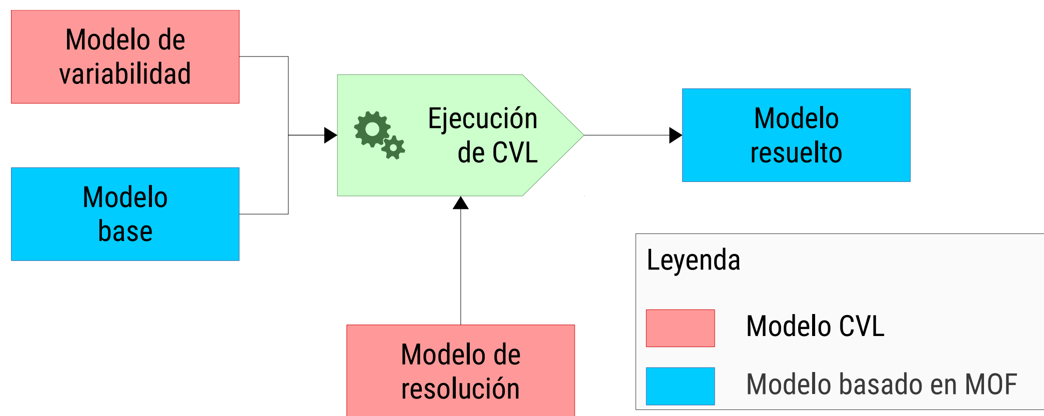


Fig. A.2 Enfoque CVL

En CVL, un modelo de variabilidad consiste principalmente de tres partes:

1. **Puntos de variabilidad**. Define los puntos del modelo base que son variables y pueden ser modificados durante la ejecución de CVL. Por ejemplo, algunos de los puntos de variabilidad soportados por CVL son la *existencia* de los elementos del modelo base o de los enlaces entre ellos, o la *asignación de valor* a un atributo.

2. **Árbol de especificación de variabilidad**. Es una estructura en forma de árbol cuyos elementos son similares a las características en FMs, y respresentan elecciones asociadas a puntos de variabilidad. Hay cuatro tipo de especificaciones de variabilidad:

    (a) *Elección*, que requiere una decisión binaria (es decir, sí/no).

    (b) *Variable*, que permite proporcionar un valor de un tipo determinado.

    (c) *Clasificador de variabilidad*, que puede ser instanciado cero o más veces, y genera un sub-árbol por cada instancia. Cada clasificador tiene una multiplicidad que indica cuántas instancias de él pueden crearse, de forma similar a los FMs con cardinalidad.

    (d) *Especificaciones de variabilidad compuestas*, que encapsulan otros árboles de especificación de variabilidad, y son usadas con propósitos de modularidad.

    Las especificaciones de variabilidad se resuelven mediante un modelo de resolución y se propagan a los puntos de variabilidad y al modelo base, generando el modelo resuelto sin variabilidad.

3. **Restricciones OCL**. CVL soporta la definición de restricciones OCL entre elementos de un árbol de especificación de variabilidad, proporcionando un mecanismo muy

flexible para delimitar los límites de la variabilidad. Estas restricciones se usan principalmente para descartar configuraciones inválidas.

A.1.2 Algoritmos genéticos

Los algoritmos genéticos (GAs) son heurísticos inspirados en el proceso evolutivo, y que son típicamente usados para encontrar soluciones en problemas de optimización. Usando GAs es posible encontrar soluciones cercanas a la óptima para problemas de optimización sin tener que explorar el espacio completo de soluciones. Como se explica en [Guo et al., 2011], aplicar GAs puede resultar muy apropiado cuando el espacio de soluciones es muy grande y no es viable evaluar todas las soluciones debido a la falta de recursos y tiempo.

En los GAs, los candidatos a ser la solución final al problema de optimización son conocidos como *cromosomas*, formando una *población*. Los cromosomas se modelan normalmente como una lista de variables binarias conocidas como *genes*, modelando cada uno de ellos una propiedad de la solución. Sin embargo, es posible definir diferentes codificaciones. La función a optimizar puede ser mono- ó multi-objetivo. En el priemr caso, la función objetivo se usa típicamente como una *función de aptitud* para medir la calidad de cada una de las soluciones, y el GA devuelve como solución al problema de optimización el individuo más apto de la población. En el segundo caso, el GA devuelve como resultado un *frente* de soluciones no dominadas. Una solución es no dominada si ninguno de los valores de las funciones objetivo puede mejorarse sin empeorar el valor de otras funciones objetivo.

Normalmente, se pueden distinguir tres estapas diferentes en la ejecución de algoritmos genéticos:

1. **Generación de la población inicial**. Se genera un conjunto inicial de soluciones para rellenar la población. El tamaño de la población es un parámetro configurable, y la elección del tamaño más apropiado depende del problema de optimización que se esté intentando resolver.

2. **Evolución a través de generaciones**. La población inicial generada en el paso anterior evoluciona para encontrar mejores soluciones. Normalmente, en cada generación, se escogen dos cromosomas y se *cruzan*, obteniendo una nueva solución que contiene genes de ambos cromosomas. Entonces, se puede introducir una *mutación* en el cromosoma resultante, cambiando el valor de uno o más de sus genes. Las mutaciones son útiles para aumentar la diversidad de la población, pero una probabilidad de mutación demasiado alta puede asemejar el problema de optimización a una búsqueda aleatoria, conduciendo a una pérdida de aptitud de la población. Al final de cada ge-

neración, el nuevo cromosoma reemplaza al menos apto de la población, mejorando de esta forma la aptitud general de la población.

3. **Devolución de la solución o frente de soluciones**. Después de la última generación, se devuelve la mejor solución o un frente de soluciones no dominadas, dependiendo del tipo de problema de optimización. Se pueden definir diferentes criterios para detener el proceso evolutivo. Por ejemplo, se puede definir, como parámetro de configuración, un número máximo de generaciones o un número máximo de evaluaciones de la función objetivo. Además, también es posible detener la evolución si el algotimo no consigue mejorar la aptitud de la población durante un número concreto de generaciones sucesivas.

A.2 Retos

Desde el punto de vista de la ingeniería, modelar aplicaciones móviles reconfigurables y desplegar las configuraciones óptimas en tiempo de ejecución es una tarea compleja. En los dispositivos móviles, los recursos son limitados y el contexto cambia continuamente. Además, el usuario espera que las aplicaciones funcionen de forma fluida y sean rápidas, así que el proceso de reconfiguración necesita ser eficiente y transparente al usuario. Por lo tanto, el desarrollo de aplicaciones móviles reconfigurables presenta una serie de retos que es necesario abordar:

1. **(R1) Habilitar los puntos de variabilidad dinámicos en tiempo de ejecución**. La primera tarea en un enfoque DSPL es modelar apropiadamente los puntos de variabilidad dinámicos, es decir, aquellos elementos que pueden ser adaptados dinámicamente. Sin embargo, estos puntos de variabilidad deben estar disponibles en tiempo de ejecución para poder generar las diferentes variantes de la DSPL. Por lo tanto, una vez se han especificado los puntos de variabilidad usando un lenguaje de variabilidad como Common Variability Language (CVL) [Object Management Group, Inc., 2012] o Modelos de Características (FMs) [Kang et al., 1990], el reto es conseguir que estén disponibles en tiempo de ejecución para poder generar las sucesivas configuraciones de la aplicación.

2. **(R2) Garantizar la consistencia de la arquitectura DSPL**. En las propuestas de DSPLs centradas en la arquitectura, el arquitecto sofware define manualmente la variabilidad arquitectónica y las restricciones entre elementos de la arquitectura, así que es posible introducir algunas inconsistencias. El reto es, por lo tanto, proporcionar al

arquitecto software herramientas para la detección automática de inconsistencias en la variabilidad, allí donde sea posible.

3. **(R3) Optimización de la configuración de la arquitectura software**. Cualquier propuesta basada en DSPLs garantiza que las sucesivas configuraciones que se generan en tiempo de ejecución son correctas con respecto al modelo de variabilidad, pero en algunas ocasiones esto no es suficiente: también debe asegurarse que estas configuraciones sean óptimas con respecto a algún criterio en concreto como, por ejemplo, las preferencias de usuario, la calidad del servicio, cantidad de recursos utilizados, etc. El principal objetivo del trabajo desarrollado en esta tesis es proporcionar un mecanismo de reconfiguración dinámica en aplicaciones que se ejecutan en dispositivos móviles, que tienen recursos más limitados. Por lo tanto, debemos considerar no sólo las configuraciones válidas sino aquellas que no exceden los recursos disponibles en el dispositivo (e.g., memoria, batería, etc).

4. **(R4) Escalabilidad del proceso de generación de configuraciones**. Se ha demostrado que el problema de la generación de configuraciones optimizadas para el contexto de ejecución, sin exceder los recursos disponibles, es NP-completo [White et al., 2009]. Por lo tanto, usar un algoritmo exacto para generar las configuraciones en tiempo de ejecución no es viable, y es necesario definir un algoritmo alternativo que garantice la eficiencia y escalabilidad de este proceso.

5. **(R5) Ejecución del servicio de reconfiguración en un dispositivo con recursos limitados**. Uno de los principales retos para cualquier servicio de reconfiguración que se ejecute en un dispositivo móvil es reducir su consumo de recursos (tiempo de ejecución, memoria, CPU, batería, etc.) tanto como sea posible. En particular, para un servicio de reconfiguración, el tiempo de ejecución resulta ser crítico ya que, para ser útil, las aplicaciones deben poder ser reconfiguradas sin que el tiempo empleado en su reconfiguración perjudique la experiencia del usuario.

A.3 Resumen de la propuesta

La propuesta presentada en esta tesis permite, por un lado, modelar aplicaciones reconfigurables (en tiempo de diseño). Por otro lado, proporciona los servicios necesarios para reconfigurar estas aplicaciones en tiempo de ejecución.

La Figura A.3 muestra una vista general del trabajo presentado. En tiempo de diseño, podemos distinguir 3 etapas diferentes en el proceso de modelado de aplicaciones reconfigurables. En la primera etapa (paso D1 en la Figura A.3), se especifica la arquitectura software,

incluyendo todos los elementos de la arquitectura (e.g., componentes y conectores). En la segunda etapa (paso D2) se especifica la variabilidad de la arquitectura software, pudiendo elegir entre dos enfoques diferentes. Por un lado, siguiendo el flujo etiquetado como D2.1 en la Figura A.3, investigado en primer lugar durante el desarrollo de esta tesis, podemos usar un perfil UML (e.g., ADOM) para modelar la variabilidad. En este caso, la variabilidad se especifica junto con la arquitectura software usando estereotipos, con la ventaja de que el lenguaje UML es ampliamente conocido en el ámbito de las líneas de producto software. Sin embargo, el principal incoveniente es que el soporte de herramientas necesario para gestionar la varibilidad arquitectónica usando perfiles UML no es tan maduro como el soporte proporcionado por herramientas para FMs. Por este motivo, proporcionamos un algoritmo de mapeo que convierte la arquitectura software con variabilidad a un FM en el que las características en el árbol representan elementos arquitectónicos, al que llamamos Modelo de Características Arquitectónico (AFM). El arquitecto software puede, de esta manera, reutilizar las herramientas existentes para FMs para analizar la variabilidad de la arquitectura software sin necesidad de manipular el AFM. En su lugar, sólo necesita interpretar los resultados obtenidos mediante las herramientas de FMs, que estarán proporcionados en términos de elementos de la arquitectura software. Así, el arquitecto software puede detectar inconsistencias en la especificación de la variabilidad de la arquitectura software y razonar sobre el grado de variabilidad. En el caso de que se detecten incosistencias, la especificación de la variabilidad puede ser refinada y analizada de nuevo, hasta que no se detecte ninguna inconsistencia.

Como alternativa, la variabilidad también puede ser especificada usando CVL (etiqueta D2.2 en la Figura A.3). En ese caso, la variabilidad se especifica en un modelo separado de la arquitectura software, lo que también presenta algunas ventajas. En primer lugar, la variabilidad se puede especificar sobre cualquier modelo que haya sido definido usando un metamodelo basado en MOF. En segundo lugar, la configuración de la aplicación obtenida una vez resuelta la variabilidad estará especificada en el mismo lenguaje empleado para modelar la arquitectura software. Por lo tanto, el arquitecto software no necesita usar un lenguaje o herramienta de diseño diferentes.

Aunque ambas alternativas son válidas para la especificación de la variabilidad de la DSPL, cada una presenta sus propias ventajas y la elección de una de ellas dependerá de los requisitos o restricciones en la especificación de la aplicación. Por ejemplo, en aquellos casos en los que que la arquitectura software no esté especificada en UML, o en los que no sea posible el uso de un perfil UML para la especificación de la variabilidad, la alternativa D2.2 podrá usarse en su lugar. Esta alternativa no sólo no requiere modificar la arquitectura software para añadir las especicifaciones de variabilidad, sino que permite usar

diferentes metamodelos para la especificacion de la arquitectura software. Por otro lado, si es posible usar perfiles UML para especificar la variabilidad, la alternativa D2.1 permite el refinamiento iterativo de la arquitectura software.

Una vez se ha especificado la variabilidad arquitectónica, la tercera etapa de nuestra propuesta (etiqueta D3) consiste en especificar la información que será utilizada para optimizar las configuraciones de las aplicaciones en tiempo de ejecución. Para este propósito, proponemos dos alternativas diferentes:

1. Definir una función de utilidad, junto con la información de uso de recursos de los diferentes elementos de la arquitectura software (e.g., consumo de batería o uso de memoria). Esta función será utilizada en tiempo de ejecución por nuestro algoritmo DAGAME para generar configuraciones que optimizan la utilidad sin exceder los recursos disponibles.

2. Definir una función multiobjetivo, con varios objetivos como usabilidad, rendimiento, uso de memoria, uso de red, etc. En este caso, se utilizará MO-DAGAME en tiempo de ejecución para generar un frente de configuraciones que optimice los valores de dichas funciones objetivo.

Una vez se han especificado la arquitectura software, la variabilidad arquitectónica y la información para la optimización, queda completada la fase de tiempo de diseño. En tiempo de ejecución, con respecto a la reconfiguración de las aplicaciones, seguimos el bucle MAPE-K de la Computación Autónoma. Concretamente, proponemos un middleware en el que un Servicio de Monitorización del Contexto (SMC) y un Servicio de Reconfiguración Dinámica (SRD) proporcionan soporte para desplegar aplicaciones reconfigurables. Las diferentes etapas del bucle MAPE-K se abordan de la siguiente forma:

1. **Conocimiento (E1)**. En nuestra propuesta, el conocimiento está compuesto por:

   (a) La arquitecta software.

   (b) La especificación de variabilidad.

   (c) La información para la optimización.

   (d) La política de reconfiguración.

   La especificación de la variabilidad será diferente de acuerdo al mecanismo de optimización elegido para especificar la variabilidad. Por ejemplo, en el caso de CVL, la información de variabilidad consiste en los puntos de variabilidad dinámicos, el árbol de *Vspecs* y las restricciones OCL. En caso de usar un perfil UML, esta información

consistirá en un AFM. La información de optimización consiste en la función de utilidad o la función multiobjetivo, y la política de reconfiguración especifica si un cambio en el contexto de ejecución debería o no iniciar el proceso de reconfiguración.

2. **Monitorización (E2)**. El SMC proporciona al SRD la información sobre la evolución de la disponibilidad de recursos, como el nivel de batería, memoria, almacenamiento, etc. Cuando se detecta un cambio, el SRD es notificado.

3. **Análisis (E3)**. Cuando se detecta un cambio en el contexto de ejecución, el SRD analiza si éste es suficientemente significativo para iniciar el proceso de reconfiguración, es decir, si los criterios de reconfiguración son satisfechos. Pueden definirse diferentes criterios para evaluar la importancia de un cambio de contexto. Por ejemplo, un cambio en el nivel de batería puede ser significativo si ha cambiado más de un 5% desde la última medida tomada, o si cambia más de un 10% por hora. Por lo tanto, se pueden definir diferentes políticas de reconfiguración, y la política aplicada es parte de la base de *Conocimiento*.

4. **Planificación (E4)**. En caso de que el analizador decida que la aplicación necesita ser adaptada, el algoritmo de optimización (DAGAME ó MO-DAGAME) es ejecutado para generar una configuración apropiada para el contexto actual. Posteriormente, se calculan las diferencias entre la coniguración actual y la nueva configuración generada, generando así el plan para cambiar a la nueva configuración a partir de la actual. Calcular las diferencias entre ambas configuraciones es sencillo, ya que pueden obtenerse mediante una operación XOR entre ellas. Tanto en el caso de CVL como en el de AFMs, las configuraciones se codifican como secuencias de unos y ceros y, por lo tanto, la generación de la arquitectura software de la configuración resultante es un proceso eficiente.

5. **Ejecución (E5)**. Por último, se ejecuta el plan de reconfiguración para adaptar la arquitectura en ejecución de la aplicación. Esto implica eliminar aquellos componentes que ya no son necesarios, añadir los nuevos componentes, conectarlos entre sí y reconfigurar los parámetros de los componentes que hayan sido modificados. Para asegurarse de que este proceso se ejecuta correctamente, todos los componentes son llevados a un estado seguro antes de ser reconfigurados. Una vez que el plan de reconfiguración ha sido ejecutado, los componentes se activan y se reanuda la ejecución del sistema.

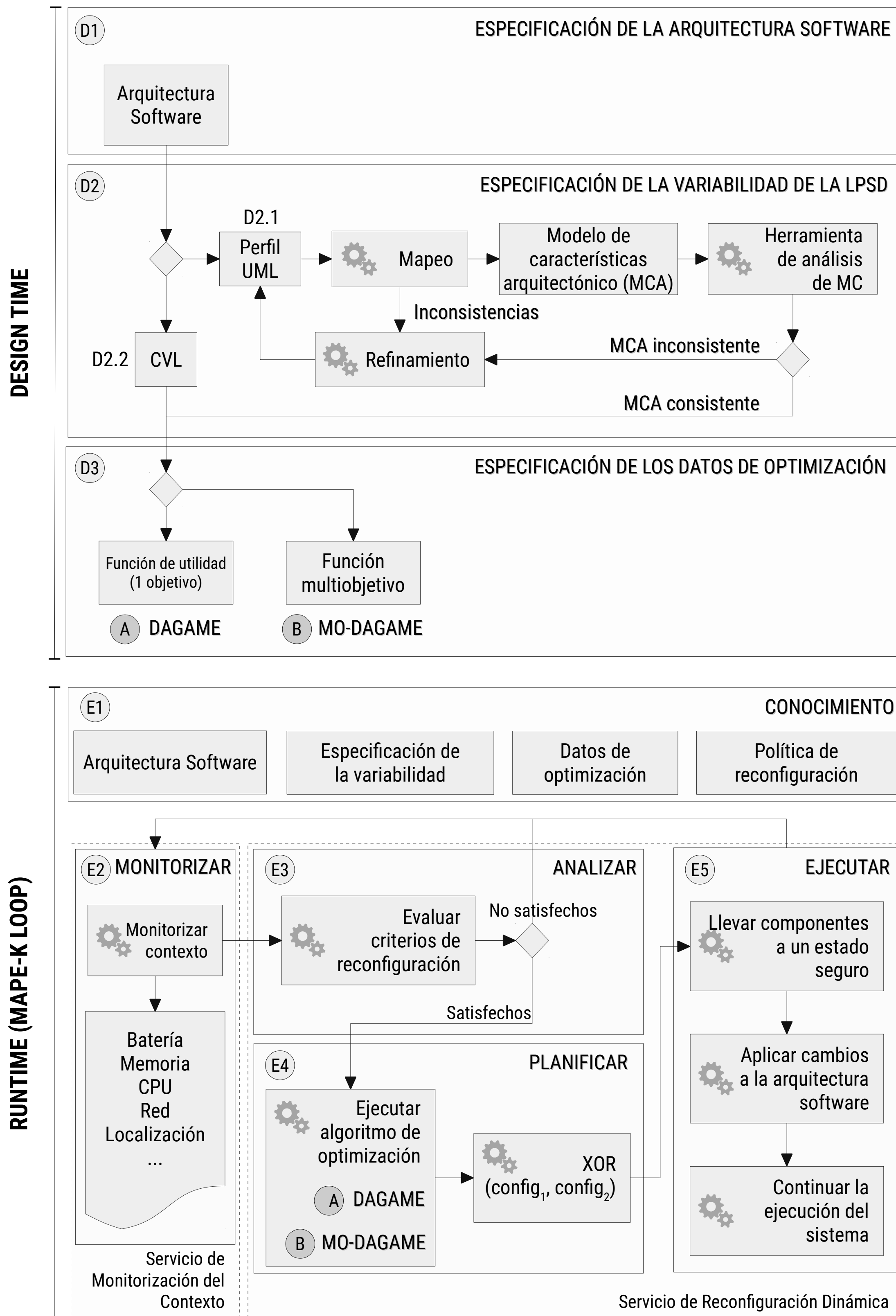


Fig. A.3 Vista general de la propuesta

A.4 Contribuciones

Como parte del trabajo desarrollado en esta tesis se ha definido una metodología para especificar aplicaciones móviles reconfigurables y una plataforma middleware para reconfigurar estas aplicaciones en tiempo de ejecución, desplegando configuraciones óptimas de acuerdo al contexto actual. A lo largo de esta sección se detallan las contribuciones de este trabajo, especificando los retos que abordan y las publicaciones en las que se incluyen.

1. **(C1) Un mecanismo para especificar la variabilidad de aplicaciones móviles usando un perfil UML y hacer que esté disponible en tiempo de ejecución**. Para este propósito, hemos definido el AFM, en el que los elmentos del árbol representan elementos arquitectónicos. Además, también hemos implementado un algoritmo que convierte el modelo de la arquitectura software a un FM que puede ser analizado en tiempo de ejecución. Esta contribución se recoge en [Pascual et al., 2013a].

2. **(C2) Definición de un proceso iterativo basado en herramientas de FM para detectar inconsistencias en la especificación de la variabilidad y razonar sobre el grado de variabilidad.** Esta contribución se presenta en [Pascual et al., 2013a].

3. **(C3) Un mecanismo para usar la información de variabilidad dinámica de la arquitectura software como entrada a un algoritmo genético**. Este mecanismo nos permite definir los servicios de reconfiguración dinámica y monitorización del contexto. Esta contribución se presenta en [Pascual et al., 2013b].

4. **(C4) Un mecanismo para especificar la variabilidad de aplicaciones móviles reconfigurables usando CVL**. De esta forma, el arquitecto software puede modelar aplicaciones reconfigurables usando el mismo lenguaje y herramientas que en el resto de aplicaciones. Esta contribución se presenta en [Pascual et al., 2015b].

5. **(C5) Definición de DAGAME, un algoritmo genético que permite la generación, en tiempo de ejecución, de configuraciones de la aplicación cercanas a la óptima, sin exceder los recursos disponibles**. Esto se consigue mediante la optimización de una función de utilidad, y ha sido implementado para la plataforma móvil Android como parte de nuestro servicio de reconfiguración dinámica. Este algotimo se presentó y evaluó en [Pascual et al., 2015b].

6. **(C6) Definición de un algoritmo genético multiobjetivo, MO-DAGAME, que genera un frente de configuraciones de la aplicación móvil optimizadas con respecto a un conjunto de funciones objetivo.** MO-DAGAME es suficientemente eficiente como para poder ser usado en dispositivos móviles. En [Pascual et al., 2015a]

se realizó un estudio para elegir el algoritmo evolutivo multiobjetivo más apropiado dependiendo de la arquitectura software y el número de objetivos a optimizar.

Tanto DAGAME como MO-DAGAME han sido evaluados aplicándolos a diferentes casos de estudio [Pascual et al., 2015a,b]. En ambos casos se han obtenido buenos resultados con respecto a los criterios del tiempo de ejecución de los algoritmos y la calidad de las soluciones generadas. Con respecto al tiempo de ejecución, hemos demostrado que pueden ser ejecutados en dispositivos móviles con recursos limitados. En primer lugar, demostramos que DAGAME genera, en menos de 22 ms, configuraciones válidas de la arquitectura software cuya utilidad es mayor al 87% de la utilidad de la configuración óptima obtenida mediante métodos exactos, que no son apropiados para dispositivos móviles. Sin embargo, dado que la calidad de las configuraciones generadas por DAGAME, así como su tiempo de ejecución, han sido evaluados mediante la simulación de la ejecución de una aplicación móvil, sería necesario llevar a cabo experimentos adicionales para determinar hasta qué punto la propuesta presentada en esta tesis es generalizable:

1. Encontrar la solución exacta al problema de optimización es un problema NP-completo y no es factible para aplicaciones móviles con un alto grado de variabilidad. Sin embargo, en el caso de aplicaciones con un número pequeño de configuraciones, podría compararse la calidad de la soluciones y el tiempo de ejecución de DAGAME con los obtenidos mediante búsqueda exhaustiva y búsqueda aleatoria.

2. Evaluar la propuesta en aplicaciones de mayor tamaño con un mayor número de configuraciones válidas de la arquitectura software.

3. Evaluar la propuesta en entornos de ejecución reales, ejecutando aplicaciones en dispositivos móviles en entornos en los que se producen cambios de contexto.

En segundo lugar, demostramos que MO-DAGAME puede ser usado en dispositivos móviles para encontrar configuraciones válidas de la arquitectura software en casos de uso reales en menos de 114 ms. Estas configuraciones se optimizaron con respecto a tres objetivos diferentes (usabilidad, consumo de batería y uso de memoria). Dado que existe un compromiso entre el tiempo de ejecución y la calidad de las soluciones obtenidas (es decir, su proximidad al frente de Pareto de las soluciones), en Pascual et al. [2015a] proporcionamos una guía para elegir el algoritmo evolutivo multiobjetivo más apropiado según el caso de uso.

A.5 Conclusiones y trabajo futuro

En este trabajo se presenta una propuesta para modelar y reconfigurar aplicaciones móviles, que cubre tanto el tiempo de diseño como el tiempo de ejecución. En tiempo de diseño, se especificacion tanto la arquitectura software, como la variabilidad de la DSPL y la información de optimización.

Se proponen dos mecanismos diferentes para modelar la DSPL. Por un lado, se puede modelar junto con la arquitectura software usando un perfil UML [Pascual et al., 2013a]. En este caso, la arquitectura software con variabilidad es mapeada automáticamente a un Modelo de Características Arquitectónico (AFM) usando el algoritmo presentado en [Pascual et al., 2013a]. Este AFM es posteriormente usando como entrada para una herramienta de análisis de modelos de características, con el objetivo de detectar inconsistencias en la definición de la variabilidad y refinar la especificación. Por otro lado, la variabilidad de la DSPL se puede modelar de forma separada de la arquitectura software usando CVL, lo que nos permite:

1. Especificar la arquitectura software usando un metamodelo basado en MOF.
2. Obtener las variantes de la aplicación con la variabilidad resuelta en el mismo lenguaje usado para especificar la arquitectura software [Pascual et al., 2015b].

Con respecto a la especificación de la información de optimización, se proponen también dos alternativas diferentes:

1. Definir una función de utilidad, junto con información del uso de recursos de los elementos de la arquictectura sofware.
2. Definir una función multiobjectivo, lo que nos permite generar un frente de configuraciones optimizadas con respecto a varios objetivos simultáneamente (e.g., rendimiento, uso de batería, uso de memoria, etc.).

En tiempo de ejecución, se propone una plataforma middleware que sigue los principios del bucle MAPE-K de la Computación Autónoma, a través de un Servicio de Monitorización del Contexto (SMC) y un Servicio de Reconfiguración Dinámica (SRD). El SMC es responsable de proporcionar al SRD información sobre los cambios en la disponibilidad de recursos como el nivel de batería o la memoria. El SRD, por otro lado, analiza los cambios en el contexto e inicia el proceso de reconfiguración si es necesario. Usando un algoritmo de optimización, genera una configuración arquitectónica adaptada al contexto

actual. Entonces, se calcula el plan de reconfiguración y se realiza la transición a la nueva configuración.

Con respecto al algoritmo para generar configuraciones arquitectónicas, se proporcionan dos opciones alternativas: DAGAME [Pascual et al., 2015b] y MO-DAGAME [Pascual et al., 2015a]. DAGAME es un algoritmo genético que genera configuraciones válidas y cercanas a la óptima, que al mismo tiempo no exceden los recursos disponibles, y todo ello lo hace optimizando una función de utilidad. Por otro lado, MO-DAGAME es un algoritmo evolutivo multiobjetivo que genera un frente de configuraciones optimizado con respecto a varios objetivos.

Esta propuesta, incluyendo los algoritmos DAGAME y MO-DAGAME, ha sido evaluada usando diferentes casos de estudios, demostrándose que genera configuraciones de la aplicación cercanas a la óptima y que es suficientemente eficiente como para poder ser aplicada en dispositivos móviles.

Como parte del trabajo futuro, hay varias líneas de investigación que podrían explorarse, como:

1. Extender el proceso de especificación de la variabilidad con soporte para la detección de inconsistencias en la variabilidad cuando se usa CVL como lenguaje de modelado de la variabilidad.

2. Evaluar más tipos de algoritmos para generar configuraciones de las aplicaciones en tiempo de ejecución.

3. Implementar y evaluar la plataforma middleware y los algoritmos de optimización en nuevos tipos de dispositivos y para más casos de uso.

References

Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., and Lahire, P. (2011). Reverse engineering architectural feature models. In *Proceedings of the 5th European Conference on Software Architecture*, ECSA'11, pages 220–235, Berlin, Heidelberg. Springer-Verlag.

Acher, M., Collet, P., Lahire, P., and France, R. (2010). Comparing approaches to implement feature model composition. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications*, ECMFA'10, pages 3–19, Berlin, Heidelberg. Springer-Verlag.

Ali, N. and Solis, C. (2015). Self-adaptation to mobile resources in service oriented architecture. In *2015 IEEE International Conference on Mobile Services*, pages 407–414. IEEE.

Ayora, C., Torres, V., Pelechano, V., and Alférez, G. H. (2012). Applying CVL to business process variability management. In *Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone*, VARY '12, pages 26–31, New York, NY, USA. ACM.

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636.

Benavides, D., Trinidad, P., and Ruiz-Cortés, A. (2005). Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 491–503, Berlin, Heidelberg. Springer-Verlag.

Blouin, A., Morin, B., Beaudoux, O., Nain, G., Albers, P., and Jézéquel, J.-M. (2011). Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 85–94, New York, NY, USA. ACM.

Brataas, G., Jiang, S., Reichle, R., and Geihs, K. (2011). Performance property prediction supporting variability for adaptive mobile systems. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 37:1–37:8, New York, NY, USA. ACM.

CAOSD Group (2009). Hydra. http://caosd.lcc.uma.es/spl/hydra/.

Casquina, J. C., Eleuterio, J. D. S., and Rubira, C. M. (2016). Adaptive deployment infrastructure for android applications. In *Dependable Computing Conference (EDCC), 2016 12th European*, pages 218–228. IEEE.

Cetina, C., Fons, J., and Pelechano, V. (2008). Applying software product lines to build autonomic pervasive systems. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 117–126, Washington, DC, USA. IEEE Computer Society.

Cheng, S.-W., Garlan, D., and Schmerl, B. (2005). Self-star properties in complex information systems. chapter Making Self-adaptation an Engineering Reality, pages 158–173. Springer-Verlag, Berlin, Heidelberg.

Cheng, S.-W., Poladian, V. V., Garlan, D., and Schmerl, B. (2009). Software engineering for self-adaptive systems. chapter Improving Architecture-Based Self-Adaptation Through Resource Prediction, pages 71–88. Springer-Verlag, Berlin, Heidelberg.

Computer Systems Group (2014). S. P. L. O. T.: Software product lines online tools. http://www.splot-research.org/.

Diaz, D. and Codognet, P. (2000). The gnu prolog system and its implementation. In *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2*, SAC '00, pages 728–732, New York, NY, USA. ACM.

Didonet, M., Fabro, D., Bézivin, J., and Valduriez, P. (2006). Weaving models with the eclipse amw plugin. In *In Eclipse Modeling Symposium, Eclipse Summit Europe*.

Djebbi, O., Salinesi, C., and Diaz, D. (2007). Deriving product line requirements: The red-pl guidance approach. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, APSEC '07, pages 494–501, Washington, DC, USA. IEEE Computer Society.

Gamez, N., Fuentes, L., and Aragüez, M. A. (2011). Autonomic computing driven by feature models and architecture in famiware. In *Proceedings of the 5th European Conference on Software Architecture*, ECSA'11, pages 164–179, Berlin, Heidelberg. Springer-Verlag.

Gomaa, H. and Hashimoto, K. (2011). Dynamic software adaptation for service-oriented product lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 35:1–35:8, New York, NY, USA. ACM.

Guo, J., Liang, J. H., Shi, K., Yang, D., Zhang, J., Czarnecki, K., Ganesh, V., and Yu, H. (2017). Smtibea: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines. *Software & Systems Modeling*.

Guo, J., White, J., Wang, G., Li, J., and Wang, Y. (2011). A genetic algorithm for optimized feature selection with resource constraints in software product lines. *J. Syst. Softw.*, 84(12):2208–2221.

Hallsteinsen, S., Hinchey, M., Park, S., and Schmid, K. (2008). Dynamic software product lines. *Computer*, 41(4):93–95.

Henard, C., Papadakis, M., Harman, M., and Traon, Y. L. (2015). Combining multi-objective search and constraint solving for configuring large software product lines. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 517–528.

IBM (2005). *Autonomic Computing White Paper — An Architectural Blueprint for Autonomic Computing*. IBM Corp.

ISA Group (2010). FaMa Tool Suite. http://www.isa.us.es/fama/.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document.

Kastner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., and Apel, S. (2009). Featureide: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 611–614, Washington, DC, USA. IEEE Computer Society.

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

Li, J., Liu, X., Wang, Y., and Guo, J. (2012). Formalizing feature selection problem in software product lines using 0-1 programming. In Wang, Y. and Li, T., editors, *Practical Applications of Intelligent Systems*, volume 124 of *Advances in Intelligent and Soft Computing*, pages 459–465. Springer Berlin Heidelberg.

Matinlassi, M. (2004). Comparison of software product line architecture design methods: Copa, fast, form, kobra and qada. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 127–136, Washington, DC, USA. IEEE Computer Society.

Object Management Group, Inc. (2002). Meta-Object Facility. http://www.omg.org/mof/.

Object Management Group, Inc. (2012). Common Variability Language. http://www.omgwiki.org/variability/.

Pascual, G. G., Lopez-Herrejon, R. E., Pinto, M., Fuentes, L., and Egyed, A. (2015a). Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications. *Journal of Systems and Software*, 103(0):392 – 411.

Pascual, G. G., Pinto, M., and Fuentes, L. (2013a). Automatic analysis of software architectures with variability. In *Safe and Secure Software Reuse*, pages 127–143. Springer.

Pascual, G. G., Pinto, M., and Fuentes, L. (2013b). Run-time adaptation of mobile applications using genetic algorithms. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, pages 73–82, Piscataway, NJ, USA. IEEE Press.

Pascual, G. G., Pinto, M., and Fuentes, L. (2015b). Self-adaptation of mobile systems driven by the common variability language. *Future Generation Comp. Syst.*, 47:127–144.

Paspallis, N. (2009). Middleware-based development of context-aware applications with reusable components. *University of Cyprus*.

Perrouin, G., Vanwormhoudt, G., Morin, B., Lahire, P., Barais, O., and Jézéquel, J.-M. (2012). Weaving variability into domain metamodels. *Softw. Syst. Model.*, 11(3):361–383.

Reinhartz-Berger, I. and Sturm, A. (2014). Comprehensibility of uml-based software product line specifications. *Empirical Softw. Engg.*, 19(3):678–713.

Rosenmüller, M., Siegmund, N., Pukall, M., and Apel, S. (2011). Tailoring dynamic software product lines. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 3–12, New York, NY, USA. ACM.

Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., and Scholz, U. (2009). Software engineering for self-adaptive systems. chapter MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments, pages 164–182. Springer-Verlag, Berlin, Heidelberg.

Sacerdoti, E. D. (1975). *A Structure for Plans and Behavior.* PhD thesis, Stanford, CA, USA. AAI7605794.

Sayyad, A. S., Menzies, T., and Ammar, H. (2013). On the value of user preferences in search-based software engineering: A case study in software product lines. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 492–501, Piscataway, NJ, USA. IEEE Press.

Shen, L., Peng, X., Liu, J., and Zhao, W. (2011). Towards feature-oriented variability reconfiguration in dynamic software product lines. In *Proceedings of the 12th International Conference on Top Productivity Through Software Reuse*, ICSR'11, pages 52–68, Berlin, Heidelberg. Springer-Verlag.

Shi, R., Guo, J., and Wang, Y. (2010). A preliminary experimental study on optimal feature selection for product derivation using knapsack approximation. In *Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on*, volume 1, pages 665–669. IEEE.

Soltani, S., Asadi, M., Gašević, D., Hatala, M., and Bagheri, E. (2012). Automated planning for feature model configuration based on functional and non-functional requirements. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 56–65, New York, NY, USA. ACM.

Trinidad, P., Cortés, A. R., Peña, J., and Benavides, D. (2007). Mapping feature models onto component models to build dynamic software product lines. In *SPLC (2)*, pages 51–56.

Tsang, E. (1993). *Foundations of constraint satisfaction*, volume 289. Academic press London.

Vassev, E., Hinchey, M., and Gaudin, B. (2012). Knowledge representation for self-adaptive behavior. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pages 113–117, New York, NY, USA. ACM.

White, J., Dougherty, B., and Schmidt, D. C. (2009). Selecting highly optimal architectural feature sets with filtered cartesian flattening. *J. Syst. Softw.*, 82(8):1268–1284.

White, J., Doughtery, B., and Schmidt, D. C. (2008). Filtered cartesian flattening: An approximation technique for optimally selecting features while adhering to resource constraints. In *SPLC (2)*, pages 209–216.

White, J., Schmidt, D. C., Wuchner, E., and Nechypurenko, A. (2007). Automating product-line variant selection for mobile devices. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 129–140, Washington, DC, USA. IEEE Computer Society.