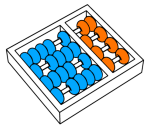
Universidade Estadual de Campinas
Instituto de Computação



Junior Cupe Casquina

A self-adaptive deployment solution for Android platforms

Uma solução de implantação auto-adaptativa para plataformas Android

CAMPINAS
2016

Junior Cupe Casquina

A self-adaptive deployment solution for Android platforms

Uma solução de implantação auto-adaptativa para plataformas Android

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientadora: Profa. Dra. Cecília Mary Fischer Rubira

Este exemplar corresponde à versão final da Dissertação defendida por Junior Cupe Casquina e orientada pela Profa. Dra. Cecília Mary Fischer Rubira.

CAMPINAS
2016

Agência(s) de fomento e nº(s) de processo(s): CNPq, 131830/2013-9

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

C92s

Cupe Casquina, Junior, 1989-
A self-adaptive deployment solution for Android platforms / Junior Cupe Casquina. – Campinas, SP : [s.n.], 2016.

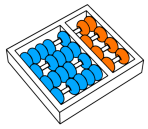
Orientador: Cecília Mary Fischer Rubira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Linhas de produto de software. 2. Componente de software. 3. Software - Arquitetura. I. Rubira, Cecília Mary Fischer,1964-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Uma solução de implantação auto-adaptativa para plataformas Android
Palavras-chave em inglês:
Software product lines
Software component
Software architecture
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Cecília Mary Fischer Rubira [Orientador]
Leonardo Pondian Tizzei
Eliane Martins
Data de defesa: 02-09-2016
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação



Junior Cupe Casquina

A self-adaptive deployment solution for Android platforms

Uma solução de implantação auto-adaptativa para plataformas Android

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira
  Instituto de Computação, UNICAMP
- Dr. Leonardo Pondian Tizzei
  IBM
- Profa. Dra. Eliane Martins
  Instituto de Computação, UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 2 de setembro de 2016

Agradecimentos

Primeiramente, agradeço a Deus por tudo o que ele faz por mim, por a minha família e por todas as pessoas do mundo. Agradeço à minha família: meus pais, meus irmãos, meus avós e meus tios porque eles sempre estiveram comigo quando tudo ia mal. Agradeço à minha orientadora, Cecília Mary Fischer Rubira, pelo apoio e pela paciência que teve comigo. Agradeço a todos meus amigos do laboratório e do mestrado. Aprendi muito com eles. Agradeço também a todos os professores do mestrado que eu pude conhecer. Eles foram uma boa fonte de experiências e conhecimentos.

Resumo

Os dispositivos móveis, hoje em dia, fornecem recursos semelhantes aos de um computador pessoal de uma década atrás, permitindo o desenvolvimento de aplicações complexas. Consequentemente, essas aplicações móveis podem exigir tolerar falhas em tempo de execução. No entanto, a maioria das aplicações móveis de hoje são implantados usando configurações estáticas, tornando difícil tolerar falhas durante a sua execução. Nós propomos uma infraestrutura de implantação auto-adaptativa para lidar com este problema. A nossa solução oferece um circuito autônomo que administra o modelo de configuração atual da aplicação usando um modelo de características dinâmico associado com o modelo arquitetônico da mesma. Em tempo de execução, de acordo com a seleção dinâmica de características, o modelo arquitetônico implantado na plataforma se re-configura para fornecer uma nova solução. Uma aplicação Android foi implementada utilizando a solução proposta, e durante sua execução, a disponibilidade de serviços foi alterada, de tal forma que sua configuração corrente foi dinamicamente alterada para tolerar a indisponibilidade dos serviços.

Abstract

Mobile devices, nowadays, provide similar capabilities as a personal computer of a decade ago, allowing the development of complex applications. Consequently, these mobile applications may require tolerating failures at runtime. However, most of the today's mobile applications are deployed using static configurations, making difficult to tolerate failure during their execution. We propose an adaptive deployment infrastructure to deal with this problem. Our solution offers an autonomic loop that manages the current configuration model of the application using a dynamic feature model associated with the architectural model. During runtime, according to the dynamic feature selection, the deployed architectural model can be modified to provide a new deployment solution. An Android application was implemented using the proposed solution, and during its execution, the services availability was altered so that its current configuration was changed dynamically in order to tolerate the unavailability of services.

List of Figures

Acronyms

AE	Architectural Element.
API	Application Programming Interface.
ART	Android Runtime.
CBD	Component-Based Development.
CBSE	Component-Based Software Engineering.
CVL	Common Variability Language.
DSL	Domain Specific Language.
DSPL	Dynamic Software product line.
FArM	Feature-Architecture Mapping.
FM	Feature Model.
FODA	Feature-Oriented Design and Analysis.
JVM	Java Virtual Machine.
MAPE-K	Monitor, Analyse, Plan, Execute, Knowledge.
MD-SPLE	Model-Driven Software Product Line Engineering.
MOF	Meta Object Facility.
NATO	North Atlantic Treaty Organization.
OMG	Object Management Group.
OOPL	Object-Oriented Programming Language.
OSGi	Open Services Gateway initiative.
PLA	Product Line Architecture.
QoS	Quality of Service.
SAS	Self-Adaptive System.
SASS	Self-Adaptive Software System.
SOA	Service-Oriented Architecture.
SPL	Software Product Line.

SPLE Software Product Line Engineering.

UI User Interface.

Contents

Chapter 1

Introduction

The increasing usage of software in current society and the whole world has prompted the creation of more Software Engineering conferences. Being the ***NATO Software Engineering Conference***, sponsored by the NATO [1], the first conference on software engineering in which experts discussed diverse issues related to this field of applied research called software engineering [114, 103].

One of the most important concepts discussed in the conference was the idea of ***software crisis*** [114]. The ***software crisis*** is a term used to refer to the main problems present in software development, either in the development process as also in the released final product [47]. Therefore, problems such as when the software development takes longer and costs more than originally estimated, or when the final released product does not function properly are included in the software crisis [47].

One way to overcome software crisis is by using what is called ***software reuse*** [40]. The idea of software reuse is not new. In 1968, Mcilroy et al. wrote "Mass-Produced Software Components", a seminal paper on software reuse [102, 40], and began to talk about the concept of software reuse. Mcilroy et al. emphasised the need to investigate mass-production techniques in software and also to consider components as if they were black boxes [102]. Black box describes what can be done, rather than how it can be done [145, 163].

Component-Based Software Engineering (CBSE) uses the software reuse in its process. The CBSE is an engineering methodology that intends to build software systems by reusable software components [87]. The interoperation, at run-time, between the binary code modules which the CBSE support, is one of the foundation blocks of our proposed infrastructure.

On the other hand, the Software Product Line (SPL) approach is one of the most promising approaches to improving the reuse of software, increasing the quality, and decreasing the time-to-market and maintenance costs [80]. In general, the basic idea of the SPL approach is to develop software products from common parts [118]. Also, most of the time, this approach uses a feature model to capture the commonalities and variability of the products. A feature model is a set of features organised along a tree-like structure which defines logical relations (optional, mandatory, alternative) among features in a

[1] North Atlantic Treaty Organization

hierarchical manner [133]. The feature model is another foundation block of our proposed infrastructure.

Software reuse can be one of the keys to developing Self-Adaptive Software Systems (SASSs) in a cost-effective manner because of this kind of systems have similar or equal components. SASSs are software systems able to self-adapt their behaviour or structure at runtime in response to changes in context [45]. Usually, a SASS uses an external adaptation approach [130] that comprises two subsystems: (i) a managed subsystem which provides the system domain functionality, and (ii) a managing subsystem which consists of the adaptation logic that controls the managed subsystem. The adaptation logic involves the implementation of an autonomic control loop, which defines how systems adapt their behaviour to keep goals controlled [108].

The MAPE-K loop [161] is a reference model to create autonomic control loops, which presents four modules : (i) a Monitor module that collects data from the managed elements, (ii) an Analyzer module that generates a report of the current situation, (iii) a Planner module that specifies what actions should be taken, and (iv) an Executor module that dispatches pre-planned actions (effectors) to modify the managed elements. The MAPE-K loop also defines that these modules exchange knowledge and data. We highlight that this MAPE-K loop is another foundation block of our proposed infrastructure.

There exists a significant number of approaches to achieve the self-adaptation properties in software systems [64]. The self-adapting systems, autonomous systems, agent-based systems, the reflective middleware, the emergent systems, etc. are all samples of these approaches. A method for developing SASSs, which borrows essential ideas from Software product line (SPL) Engineering, is called Dynamic software product lines (DSPLs) [49, 143]. An SPL is a set of software-intensive systems that share a common and 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 [33].

In particular, A DSPL [63] is an SPL that can be reconfigured at runtime and it emphasises variability analysis and design at development time and variability binding and reconfiguration at runtime. The DSPL concepts are another foundation block of our proposed infrastructure.

1.1 The context

Mobile applications are changing the way that we see the today's world because several of them are useful tools for everyday activities [24]. Moreover, smartphones, nowadays, provide the same computing power and similar capabilities as personal computers of a decade ago [24], allowing the development of sophisticated mobile applications. Consequently, nowadays mobile applications may require several non-functional requirements [31] such as reliability, robustness, distributed execution, and the ability to deal with variations in services availability and computing resources.

However, most of the today's mobile applications are not able to achieve non-functional requirements such as the robustness and fault tolerance because most of these are unable to tolerate or cope failures, errors or faults during their execution. Robustness [1] is a

non-functional requirement that permits software systems to remain viable in the presence of both disturbances from within the system itself as well as those from its environment. Fault tolerance [12] is a non-functional requirement that requires a system to continue to operate, even in the presence of faults.

Dynamic deployment is the process to update, patch or extend features or components without stopping the application [122]. Software systems may use the dynamic deployment to tolerate or cope failures, errors or faults during their execution. Nevertheless, today's mobile applications are deployed using static configurations. A deployed application with a static configuration is ready for an initial context and, in general, it is not ready for changes that may occur during runtime. In contrast, dynamic deployment allows applications to self-adapt to changes that happen in the environment.

Self-adapt to changes that occur in the environment is an essential feature of the self-adaptive systems (SASs) and hence of the SASSs. Over the years, the academic literature has presented a wide variety of approaches with the sole purpose of developing SASs. Among them are [85] model-based approaches, architecture-based approaches, service-oriented approaches, agent-based approaches, nature-inspired approaches, approaches that use the control theory, and more. Many of these approaches also are used to develop SASSs [27, 112].

However, despite the existence of many studies, building SASSs on Mobile platforms (Android, Symbian, iOS, Blackberry, and so on) is not an easy task because the lack of infrastructures and tools that facilitate the development of this kind of systems.

1.2 Definition of the problem

Problem - Little support that exists in mobile platforms to build self-adaptive software

A diversity of approaches have been proposed for the development of self-adaptive software [97] such as Rainbow, Archstudio, MUSIC, and so on. However, many of the implementations of these methods are not applied to the development of self-adaptive software on Android devices.

For example, implementations based on the ***JBoss AOP*** framework such as the proposal of Shen et al. [144] are not support on the Android platform because this framework does not run on this platform. Even being the ***JBoss AOP*** a Java framework [74] for programming aspect-oriented applications [68] that can be used in any programming environment [8].

Also, many of these implementations are complicated and hard to use and understand. The MUSIC middleware [128], for example, runs on an OSGi framework. An OSGi framework is a Java framework that runs on the top of a Java environment, and it is an implementation of the OSGi specification [93]. The OSGi specification is a collection of standard application programming interfaces (APIs) that define a service gateway [55].

The MUSIC middleware can be executed on mobile devices. However, running an OSGi framework on a mobile device is not an easy task because of the continuous evolution

of the devices. Moreover, the authors of MUSIC middleware do not offer a mobile version to facilitate the development of this kind of systems.

To conduct our study, we define the following research questions:

- **Research Question 1 (RQ 1) :** How can a software system achieve a self-adaptive behaviour on the Android platform?
- **Research Question 2 (RQ 2) :** By getting a solution to develop self-adaptive applications on the Android platform, what is the performance of the proposed solution?

As far as we know, building self-adaptive software in a cost-effective and predictable manner is a challenge [97] because the lack of tools and simple solutions to the development of this kind of system.

1.3 General view of the proposed solution

Proposed solution - A deployment infrastructure to support the development of self-adaptive software on the Android platform

We have proposed an infrastructure to the development of SASSs on Android platforms based on queries, at runtime, to variability model and dynamic SPL architecture. We decided to use a feature model as the variability model of the system because each product can be differentiated from the others by means of its features and each feature is a characteristic of a product that is visible to the end-user in some way [126].

In addition, our proposed solution follows the IBM's MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) reference model for autonomic control loops [71] because this model is used (consciously or unconsciously) in some way for many academic studies. Therefore, in the academic literature we find several infrastructures and frameworks which use as reference the IBM's MAPE-K Loop, including Rainbow [52], StarMX [6], ASF [57], FUSION [45], ArCMAPE [113] and so on. Figure 1.1 shows the MAPE-K Loop proposed by IBM in which the Autonomic Manager manages the Managed Element through the effectors.

Additionally, we have implemented a reusable Java framework that supports the development of SASSs using our infrastructure. We have used our framework to add self-adaptive properties to an Android application called Buscame. Our proposed solution for Android applications implements the IBM' MAPE-K loop and responses to the **RQ 1**. We highlight that in this work we have used the ideas behind the dynamic software product lines (DSPLs) approach to propose and implement our infrastructure and framework.

Finally, we measured the performance of the proposed solution to response the **RQ 2**.

1.4 Related work

In the context of Android applications, Maly and Kriz [100] present two implementation details to support dynamic deployment of new functionalities on an Android application.

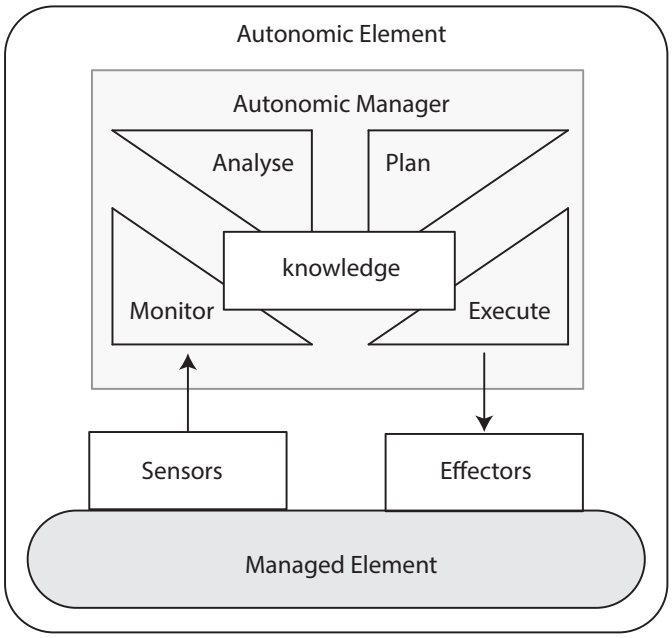


Figure 1.1: IBM's MAPE-K reference model for autonomic control loops

The first solution using the Java Reflection API to load and launch activities, and the second using modules based on Android fragments [2]. However, these solutions just explain how to manage user interfaces and not business logic.

Moreover, in academic literature, we found many works applying different types of approaches and frameworks to develop SASSs. Among them, Krupitzer et al. [85] and Macías-Escrivá et al. [97] collected and organised several of these approaches. Some of these showed approaches are the model-based approaches, architecture-based approaches, service-oriented approaches, and so on.

Rainbow [52], ArcMAPE [113] and MoRE [28] are solutions that also use the MAPE-K loop [161] as our proposed solution.

1.4.1 Rainbow

Rainbow is an architecture-based platform for self-adaptation, which provides reusable components that we can use to create other SASs [52]. Rainbow provides an external adaption approach to self-adaptive systems and, in addition, has mechanisms for monitoring a target system and its environment, detecting opportunities for improving the system's quality of services (QoSs), deciding the best course of adaptation based on the state of the system, and effecting the appropriate changes through system-level effectors [37].

In the Rainbow framework, the Model Manager component updates the architecture model using the information observed in the system via probes [29, 52], the Strategy Executor component executes the chosen strategy on the running system via system-level effectors [37, 52], the Adaptation Manager component chooses a suitable adaptation strategy based on current state of the system (reflected in the model) [29, 37, 52], and the Architecture Evaluator component evaluates a set of constraints defined on the architec-

ture model to ensure that the system is operating within an acceptable range [29, 37, 52].

We highlight that Rainbow framework has demonstrated that the models (updated at run-time) belong to a system can form the basis to detect and correct (at run-time) several problems in a system [164]. We rescued this idea to strengthen our framework.

1.4.2 ArcMAPE

Nascimento et al. [113] presents ArCMAPE, a self-adaptive infrastructure to instantiate appropriate fault tolerance techniques at run-time in response to the context changes. ArCMAPE used mainly three core elements [113].

- A feature model to model the fault tolerance techniques applied to service-oriented architectures. Service-oriented architecture (SOA) is an architectural style whose goal is to achieve loose coupling among interacting software agents [123].
- A product line architecture (PLA) of fault tolerance techniques also applied to service-oriented architectures.
- The common variability language (CVL) to model the variability. CVL [165] is a generic variability modeling language that can be applied to models created in any Domain Specific Language (DSL) that is defined based on Meta Object Facility (MOF). MOF is a standard representation for meta-models and models proposed by OMG (Object Management Group) [43], a technology consortium formed in 1989 [23].

To use this infrastructure, we firstly have to generate all the valid states (models) of the software using the SINTEF' tool which implement the CVL. Then at run-time in response to the context changes the infrastructure uses an appropriate model (generated at design time) to upload and load the bundles associated with the model using the OSGi framework. The authors also suggested applying the FArM method (Feature-Architecture Mapping) to improve the mapping between features and the PLA elements of the system.

This infrastructure presents two main disadvantages: (i) the infrastructure uses an internal adaptation approach because it mixed the managed subsystem with the managing subsystem and (ii) the infrastructure has a mocked implementation.

1.4.3 MoRE

Cetina et al. [28] presents MoRE, a model-based reconfiguration engine to implement model management operations. MoRE uses the feature model and the OSGi framework to implement their solution. In general, using a variability model, the context and the Domain Specific Language model, MoRE generates a reconfiguration plan which later is used to modify the system using the OSGi API.

MoRE is more closely related work to our framework, because MoRE uses a feature model as the base of the self-adaptive system and also, it uses a set of registers which the current state of a sensor (context monitor) as in our infrastructure.

The main disadvantages presented in MoRE is the use of the OSGi framework to manage the bundles of the system at run-time because the OSGi framework, for now, is not easily used on Android platforms. Many times only finding a suitable documentation to run the OSGi framework in a particular Android platform is a nightmare.

1.4.4 Other solutions

The FUSION framework is also used to develop SASs which combine feature-orientation, learning, and dynamic optimization [45]. FUSION mainly uses the online learning to analyse and self-tune the adaptive behaviour of the system to unanticipated changes [45].

EUREMA is a model-driven approach used to develop self-adaptive software [155]. EUREMA provides a domain-specific modelling language to specify the feedback loops and also an interpreter to execute this specification [155].

Another framework to develop SASSs is proposed in [111]. Also, the authors present a system integrity verification method (policy-based). Moreover, Ding et al. [42] suggested the use of the Petri nets with neural networks to model self-adaptive software systems.

1.5 Contributions

Next, we summarise the main contributions of our research work.

- **Contribution 1 - A new deployment infrastructure to support the development of self-adaptive software**

  The new infrastructure and its implementation received the name of Cosmapek. It is one of the first infrastructures that works on Android platforms. In particular, Cosmapek can be used on devices that have a Java Virtual Machine (JVM). Cosmapek joined several ideas of different research areas such as DSPL and SAS in order to get the expected result.

- **Contribution 2 - A new self-adaptive software that runs on the Android platform**

  The new self-adaptive software, called Buscame, is an Android application. It is one of the first self-adaptive applications available to this kind of platforms. In particular, Buscame was tested on the Lollipop platform. The app internally uses the Cosmapek to get the self-adaptive behaviour. .

1.6 Outline

The roadmap consists of four chapters.

- **Chapter 2**, *Background.* The chapter presents the background related to our work. We begin examining the concepts refer to the software product lines and ending exploring concepts refer to the self-adaptive systems.

- **Chapter 3**, *The Cosmapek Infrastructure.* The chapter presents the Comapek infrastructure and its framework. Also, we present a new component implementation model called Dicosmos.

- **Chapter 4**, *The Cosmapek Infrastructure Applied to Android.* The chapter presents our self-adaptive application, which is an Android application. We also present the details of implementation of it along with some performance tests.

- **Chapter 5**, *Conclusions and Future work.* The chapter presents the conclusions and contributions of our work.

1.7 Final remarks

In this chapter, we have presented the problem to be solved, an overview of our proposed solution and the related works to our solution. First, we introduced the context of our problem. Self-adaptive software on mobile devices is our context. Second, then of presents the issue, we proposed a solution which mixes the ideas of the SASSs, DSPLs and CBSE. Finally, we showed the related work and the contributions of the thesis.

Chapter 2

Background

This chapter shows most of the required background to understand our work.

Our solution proposed in the Chapter 3 mainly is based on the dynamic software product lines (DSPLs) approach. To understand better what a DSPL means, first, we need to explain the concepts that gave rise to the DSPLs approach. Among these are the product line, the software product line and the software product line engineering.

Our solution uses a feature model at runtime to manage the variability. To understand better what is this model we need to know what is a variability model at design and runtime time. Our solution also allows creating a self-adaptive software using architectural reconfigurations. In this chapter, we review these concepts.

The self-adaptive application in Chapter 4 has a service-oriented architecture (SOA). Besides, it is deployed on an Android platform. Therefore, we show some notions about SOA and the Android architecture.

2.1 Software product lines

2.1.1 Product lines

A product line, also known as a product family [86] or a portfolio product [58], is a set of similar products, which were built using common and different components [34, 127, 159, 90]. Being the components also known as assets. Figure 2.1 shows an example of a product line of devices in which each device shares common features with the other devices.

Similarly, product lines, also known as product families or product line groupings [138], are sets of products that share common features [154, 147]. Examples of product lines are manifold and can be found in different domains [126]. For example, in the market for mobile phones, we found several types of phones with shared and different features and in various models [126]. Additionally, we can even select the external and internal features that we want for our mobile phone, and obtain a product with this feature configuration.

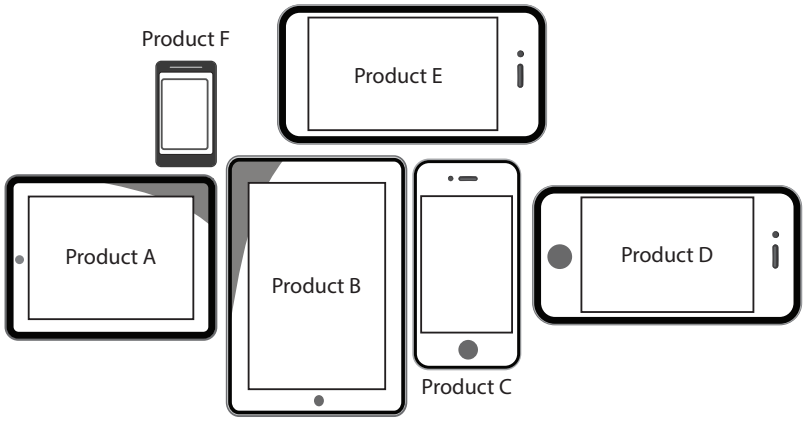


Figure 2.1: A product line of devices

2.1.2 Software product lines

A software product line (SPL) is a set, family or collection of software intensive systems, software intensive products, software systems, systems, or programs that have common and variable features [115, 148, 151, 59, 77, 129, 120]. An SPL, besides, is developed from a common set of core assets in a prescribed way [126]. Many samples of SPL are found in the real world. A good sample of an SPL is the set of web browsers of the market. The browsers are software programs that share common features and theoretically can be built from common components. Figure 2.2 shows an SPL of browsers in which a set of different components generates a different browser.

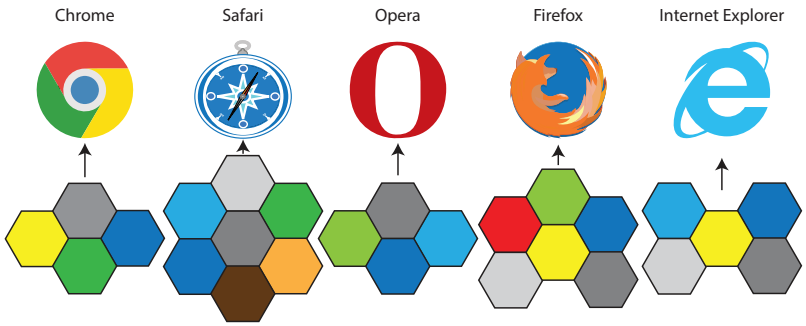


Figure 2.2: A Software Product Line of browsers

Software product lines (SPLs) are families of software systems that shares features, built by reusing software assets abstractly represented by features [67, 35, 107].

2.1.3 Software product line architecture

The software architecture is critical among the collection of core assets that form the basis of a family of software systems or families of software systems [115]. Core assets often include, but are not limited to, test plans, performance models, domain models, test cases, and so on [115]. A software product line (SPL) architecture represents the whole software structure of all potential products that can be generated by one or more software product lines [76]. An SPL architecture also describes the commonality and variability [110] of the set of goods. In the literature, we can find different SPL architectures for

diverse domains such as the SPL architecture presented by Waku et al. for data collection domain [156]. Figure 2.3 shows a component diagram (in UML 2.0 notation) of one of the architectures of an SPL called MobileMedia [149].

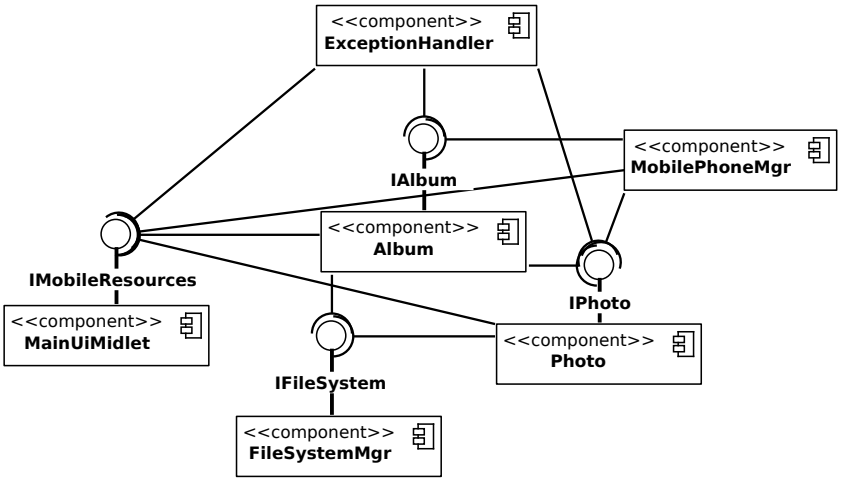


Figure 2.3: Architecture of the MobileMedia SPL [149]

2.1.4 Software product line engineering

Software product line engineering (SPLE) provides concepts on how to develop software product lines (SPLs) [21]. In particular, the SPLE is a paradigm or approach to develop software products [26, 94, 139] using software platforms and mass customization [124, 46, 134]. A software platform is a set of software subsystems and interfaces or generic components that form a common structure, from which a set of software products can be efficiently developed and produced [124, 105, 46]. Mass customization is the large-scale production of goods tailored to individual customers' needs [124].

SPLE paradigm habitually separates the developing of software products in two processes: the domain engineering and application engineering [124]. The domain engineering is a process in which the commonality and the variability of the product line are defined and realised [124]. On the other hand, the application engineering is a process in which the products are built by reusing domain artefacts and exploiting the product line variability [124]. Domain artefacts, also called product line artefacts, are reusable development artefacts created in subprocesses of domain engineering. A development artefact is an output of a subprocess of the domain or application engineering [124]. These development artefacts encompass requirements, architecture, components and tests. [124].

An SPLE framework is an abstract representation of the two processes defined earlier for developing software products along with their produced assets. An asset is a result of the development process. It can be more tangible as data, design and software code, or even more intangible, such as knowledge and methodologies [140]. In the literature, many methods and frameworks have been presented for creating software product lines Figure 2.4 shows one of these SPLE frameworks in which we can clearly distinguish the two processes as mentioned earlier.

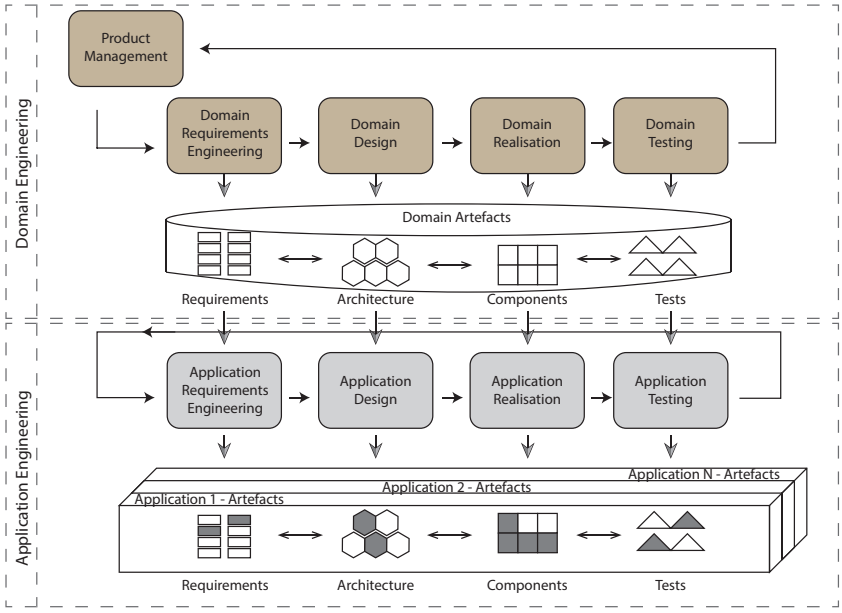


Figure 2.4: A software product line engineering framework [124]

An interesting paradigm or approach that is generating attention now is the Model-driven Software Product Line Engineering (MD-SPLE), which combines the model-driven software development with the SPLE [36]. The model-driven development is a development paradigm that uses the models as a primary artefact of the development process [22].

2.2 Software variability

Variability is the ability of a system to be changed, customised or configured so that it fits or reuses in a particular context [152, 48]. Variability is applied to software systems or software artefacts (e.g., component) [48]. Thus, it allows us to adapt the structure, behaviour, or underlying process of the software systems [48]. Variability, besides, is a fundamental concept in software product lines and facilitates the development of different versions of a software systems or software artefact [48].

Mainly we found two types of variability: the **dynamic variability** and the **static variability**. When the variability is determined at runtime, we are talking about dynamic variability, and when the variability is decided at design or development time, we are talking about static variability (or design-time variability) [48]. In principle, dynamic variability is much more complicated than conventional static variability [4].

The variability in the software architecture is usually introduced through variation points (i.e., locations where change may occur), allowing the development of a different version of a software architecture [99]. The dynamic variability in the software architecture is a fundamental concept that we took to propose our infrastructure (See Chapter 3)

2.2.1 Variability models

A model is a simplified or partial representation of reality [22]. In particular, the modelling is essential to human activity because every action is preceded by the construction implicit or explicit of a model [25].

Variability Models are designed for modelling the variability of something. In the context of software, a variability model is a model that defines the variability of a software [84]. We found different variability models in the literature. Some of the more common variability models are [60] the goal model, the feature model, the unified modelling language, the orthogonal variability model, the common variability model. Whereas the orthogonal variability model and the feature model are the classic models that define the variability of a software product line [124].

Variability languages also describe the variability of a system [73, 16]. The Kconfig variability language, for example, was created to describe the variability of the Linux kernel [16]. In particular, the variability models and languages define, document and manage the commonalities and variability of reusable artefacts [135, 16] such as software components, requirements, test cases, etc.

2.2.2 Feature model

A feature model describes the information of all possible products of a software product line regarding features and relationships among them [13]. In literature different feature model variants were presented [136]. The feature model proposed by the Feature-Oriented Design and Analysis (FODA) method is the most traditional variability model in the academic literature and the market [10].

To analyse a feature model, we can use the proposition logic. Tools like SAT solver, Alloy, BDD solver and SMV were created to examine an feature model using this proposition logic [13]. In the implementation of our framework, we use the SAT solver. The core notion in feature models is the features [5] in which a feature is classified as a mandatory, optional, OR, or XOR feature. Besides, each kind of feature defines rules.

- Mandatory feature: if the parent feature is selected, the mandatory child features must also be selected in the feature configuration.
- Optional feature: if the parent feature is selected, the optional child feature may or may not be selected in the feature configuration.
- OR feature: if the parent feature is selected, one or more features in the OR feature group must be selected in the feature configuration.
- XOR feature (Alternative feature): if the parent feature is selected, one and only one of the features in the XOR feature group must be selected in the feature configuration.

A sample of a feature model (in Czarnecki-Eisenecker notation) of an SPL for the chess domain [157] is shown in the figure 2.5.

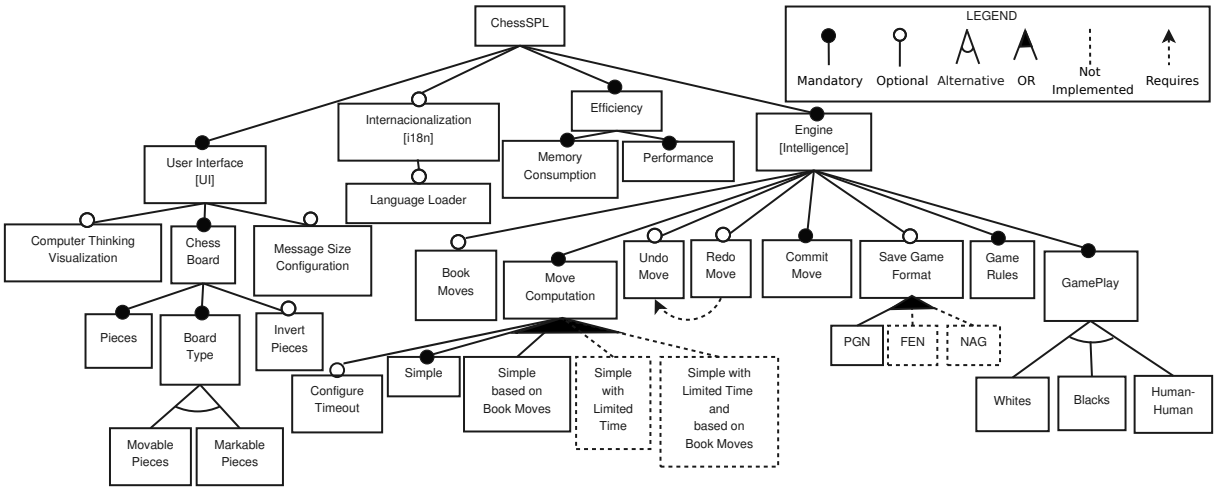


Figure 2.5: A feature model of a SPL for chess domain [157]

2.2.3 Variability models at runtime

Software models [15] have the potential to be used at runtime either to monitor and verify particular aspects of runtime behaviour as well as to implement self-*capabilities (e.g., self-managing, self-healing). It is because software models can be used to manage all or partially all of the possible configurations of a SASS in the same way as a variability model.

A variability model stores information about the variability of a system. Each variability model element can be declared dynamic or static at design and runtime time. The dynamic elements can or cannot be taken into account; however, the static elements always have to be considered. Moreover, queries, at runtime, to this kind of model assist in achieving the self-adaptive or autonomic behaviour of a self-adaptive software system (SASS) [28].

The management of adaptation at runtime using models at runtime is proper of the Models@runtime approach. The Models@runtime approach [88] provides higher levels of abstraction of both the running systems and its environment.

2.3 Self-adaptation

The self-adaptation is an essential feature of natural evolution [41] and also a property at the top of a complex self-* taxonomy [19]. The self-* properties [131], also known as adaptivity properties, are grouped in a hierarchy of three levels: the general level, the major level, and the primitive level. Many of these self-* properties are similar to each other. Figure 2.6 shows the list of Self-* properties presented in [131].

2.3.1 Self-* properties

- **Self-awareness** is a property which the system is capable of knowing about itself i.e. its states and behaviours [18, 70, 131].

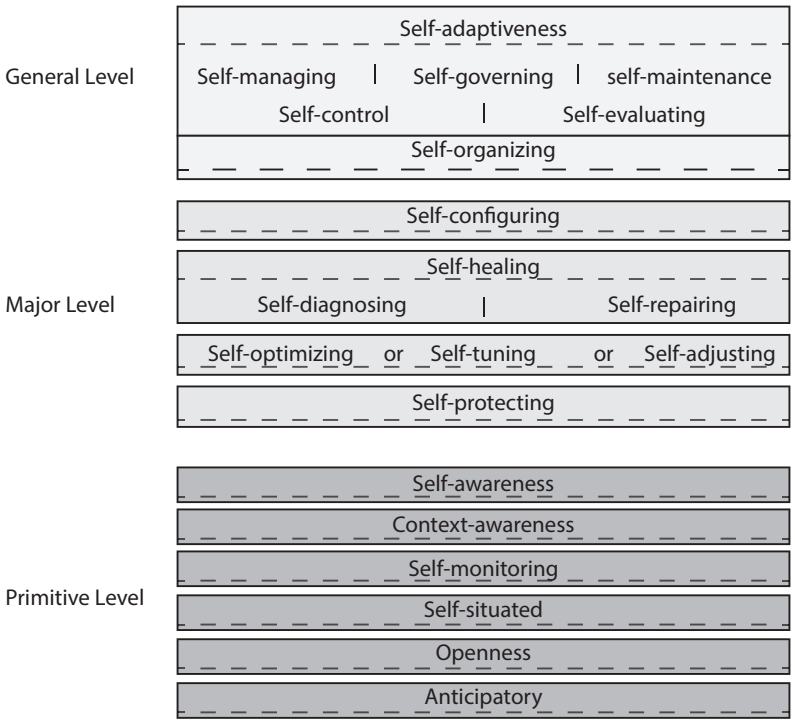


Figure 2.6: Self-* properties

- **Context-Awareness** is a property in which the system is aware of its execution environment and, in addition, is able to react to changes in its environment [119, 131].
- **Self-monitoring** is a property which the system is able to detect changing circumstances [70].
- **Self-situated** is a property which the system is aware of the current external operating conditions [70].
- **Openness** is a property which the system should function in a heterogeneous world. Thus, it should be portable across multiple hardware and software architectures [119]. Additionally, it must be built on standard and open protocols and interfaces.
- **Anticipatory** is a property which the system should be able to anticipate, to the extent possible, its needs and behaviours when a stimulus is manifested. Additionally, it has to be able to manage itself proactively [119, 72].
- **Self-protecting** is a property which the system should be capable of detecting and protecting its resources from both internal and external attack. Also, it has been able to maintain the overall system security and integrity[119].
- **Self-optimizing**, also known as self-tuning or self-adjusting, is a property in which the system manages the performance and the resource allocation to improve its execution [70, 119, 131]. Also, it has to react to the user's policy changes within the system [70]. End-to-end response time, throughput, utilization, and workload are examples of important concerns related to this property [131].

- **Self-repairing** is a property in which the system focuses on recovery from errors, faults, and failures [131].
- **Self-diagnosing** is a property in which the system focuses on diagnosing errors, faults, and failures [131].
- **Self-healing** is a property in which the system has the capability of discovering, diagnosing, and reacting to disruptions [131, 70, 119]. It also monitors vital signs to predict, avoids and prevents health problems, and the undesirable levels [131, 70].
- **Self-configuring** is a property in which the system is able to readjust itself automatically in response to changes by installing, updating, integrating, and composing/decomposing of software entities [70, 131].
- **Self-organizing** is a property in which the system is able to change their internal structure and their function in response to external circumstances [11].
- **Self-evaluating** is a property in which the system is able to evaluate their innate characteristics [89].
- **Self-control** is a property in which the system is able to control their innate characteristics [83].
- **Self-maintenance** is a property in which the system is able to repair or maintenance itself in response to change in the environment [62].
- **Self-governing** is a property which the system has the ability to decide and implement decisions for and by oneself [61].
- **Self-managing** is a property which the system is able to free from the details of system operation and maintenance and, in addition, is able to provide to users with a system that runs at peak performance 24/7 [79].
- **Self-adaptiveness** is the property (at the top of a complex self-* taxonomy) which comprises the self-evaluating, self-control, self-maintenance, self-maintenance, self-governing and self-managing properties [131].

2.3.2 Autonomic systems

An autonomic system is a collection of autonomic elements [30]. Autonomic applications and systems are capable of managing their behaviours and their relationships with other systems/applications by policies. Each Autonomic system can be composed of several autonomic elements [119].

An autonomic element is an individual system that contains resources and delivers services to other autonomic elements [79]. In particular, in the autonomic systems, each autonomic element manages their internal behaviour and their relationships with others by policies that have been established in the environment [79]. An autonomic element

sometimes is called an autonomic system. Figure 2.7 shows an autonomic element interacting with other autonomic elements. We highlight that each autonomic element follows the IBM's MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) reference model.

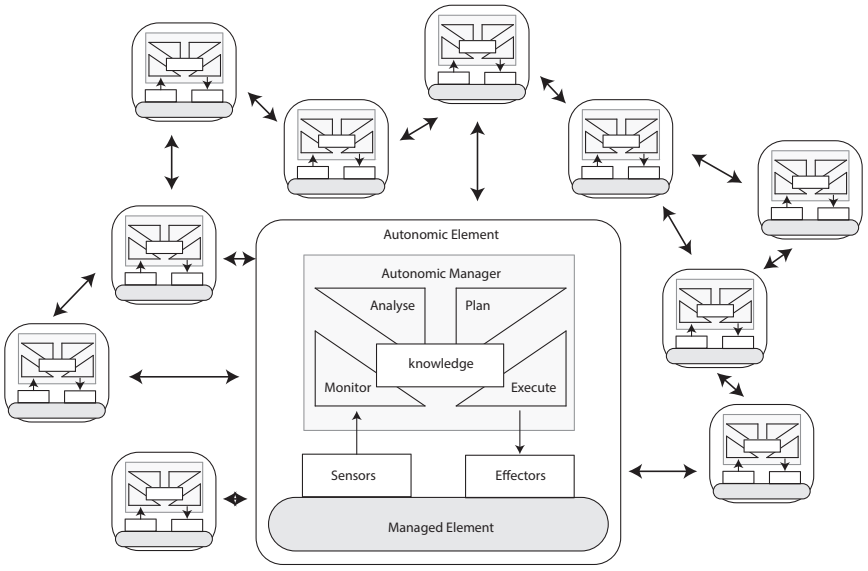


Figure 2.7: An autonomic element interacting with other autonomic elements

In the software domain, the autonomic systems are also called autonomic software systems. In particular, each autonomic element of the autonomic software systems is an autonomic software system or autonomic software [81]. In other words, an autonomic element is an autonomic system. Some of the features that form an autonomic software are defined by [51]. These features, also, have a strong relation with the above self-* properties.

- An autonomic software system needs to know itself
- An autonomic software system must configure and reconfigure itself under unpredictable conditions.
- An autonomic software system looks for ways to optimise its work.
- An autonomic software system must be able to recover from events that might cause malfunctions.
- An autonomic software system must be an expert in self-protection.
- An autonomic software system knows its environment and the context surrounding its activity. Also, it acts accordingly.
- An autonomic software system cannot exist in a hermetic environment (and must adhere to open standards).
- An autonomic software system has to anticipate the optimised resources needed to meet the user's information needs while keeping its complexity hidden.

The self-* properties frequently specify the expected behaviour of autonomic software; these properties play a significant role in determining the target goal [132].

2.3.3 Self-adaptive systems

The Self-Adaptive Systems (SAS) have been used and studied in different areas of research [97, 131], including software engineering, artificial intelligence and so on. As in autonomic software, the self-*properties [132] frequently specifies the expected behaviour of self-adaptive systems.

In the domain of software, these self-adaptive systems are also called self-adaptive software systems, self-adaptive application or simply self-adaptive software [131, 39]. A self-adaptive software is a kind of software that disposes of adaptation mechanisms that allow continued operation when changes exist in its operating environment [146, 85]. In other words, it is a software that modifies its behaviour in response to changes in its environment [117] to continue to operate.

The environment changes refer to anything observable by the software through end-user input, external hardware device and sensors. In particular, this kind of systems has two types of sensors: a physical sensor that is a device which responses to stimuli by generating of processable outputs [75] and a virtual sensor that is an emulation of a physical sensor which obtains its data using physical sensors [98].

Many approaches for developing self-adaptive systems are exhibited in scholarly literature. An overview of these approaches is presented in [85]. The Rainbow framework, Archstudio, Architectural run-time configuration manage, and 3L approach, for example, use an architecture-based approach for developing self-adaptive systems [85]. Dynamic Software product lines and MUSIC use a Model-based approach for developing self-adaptive systems [85]. Autonomic Computing, Autonomic Communication, Control loop UML profile, Control loop patterns, and Control theory foundation use an approach based on control theory [85]. Moreover, all these approaches share common and different ideas.

2.3.4 Dynamic software product lines

Dynamic Software Product Lines (DSPLs) is an approach which applies ideas developed in the SPL community to build software that adapts dynamically at run-time [63, 64]. The variability model is the central artefact in the SPLs engineering and in the DSPLs approach for specifying the commonalities and variability [121]. In DSPLs, the variability model describes the potential range of variations that can be produced at run-time for a single product [121, 14]. Where a dynamic feature can be (de)activated at runtime while a static feature is mandatory and cannot be deactivated at runtime [153]. We use these core ideas in our framework and infrastructure.

2.3.5 External/Internal adaptation

The adaptation approaches in the self-adaptive systems can be divided into two categories: An external adaptation approach and internal adaptation approach. Figure 2.8 shows the internal and external approaches to develop a self-adaptive software.

When a self-adaptive software system uses an internal adaptation approach, the whole set of sensors, effectors, and adaptation processes are mixed with the application code

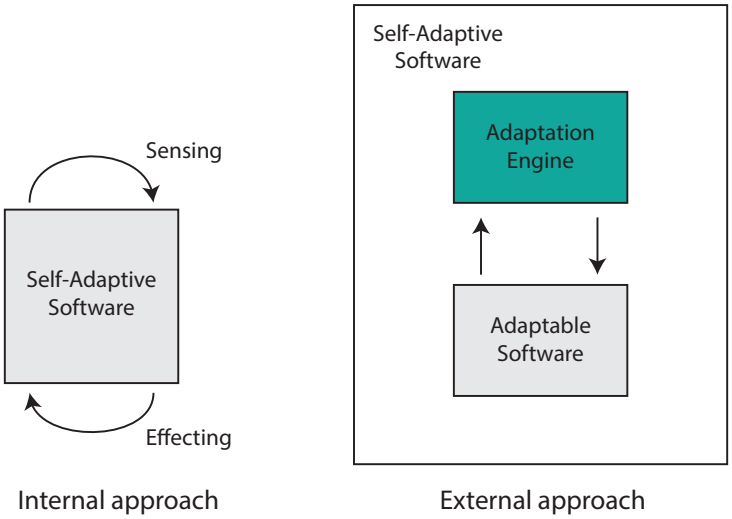


Figure 2.8: Internal and external approaches for developing of self-adaptive softwares

[131]. Then this approach offers a poor scalability and maintainability [131]. On the other hand, when a self-adaptive software system uses an external adaptation approach, it splits the self-adaptive software systems in two: the adaptation engine and the adaptable software [155]. Then this approach is the ideal approach to follow. We use this approach in our framework and infrastructure.

2.4 Architectural reconfiguration

An architectural configuration is a connected graph of components and connectors that describes the architectural structure [160]. Architectural reconfiguration consist in modifying this structure by adding or removing components or connectors [95]. In particular, components and connectors are architectural elements [125] that can be created using an object-oriented programming language (OOPL) in a variety of ways; Even, it is possible to build these without using an OOPL [9].

A component architecture is a description of a system in form of a collection of components that interact with each other through connectors [44, 7]. Moreover, components are the foundation for the software product line (SPL) development [20], the component-based software engineering (CBSE) and the component-based development (CBD). Component-based development [69] is a software development approach in which all aspects and phases of the development lifecycle, including requirements analysis, design, construction, testing, deployment and project management, are based on components.

To develop self-adaptive software systems (SASSs) we require adaptations at the architectural level at runtime. Adaptations at the architectural level at run-time require of changes, on the fly, between the components and their connectors (also known as interconnections) [117].

2.4.1 OSGi

OSGi is a component framework specification that brings modularity to the environment [158]. OSGi is not new. It has been around since the late 1990s [82], allowing us to create dynamic architectures [56] at runtime from that time. The OSGi specification has several implementations (Apache Felix, Knopflerfish, Eclipse Equinox, so on) that work only in Java environments, but exists some other implementations, for example, nOsgi, which works with C++.

The modular entity in the OSGi component framework specification is known as a bundle. A bundle is a collection of code, resources, and configuration files which are packaged in a Java Archive (JAR) [54]. The OSGi specification also defines that each bundle has to be the same life cycle. Figure 2.9 shows the life cycle of a bundle in OSGi.

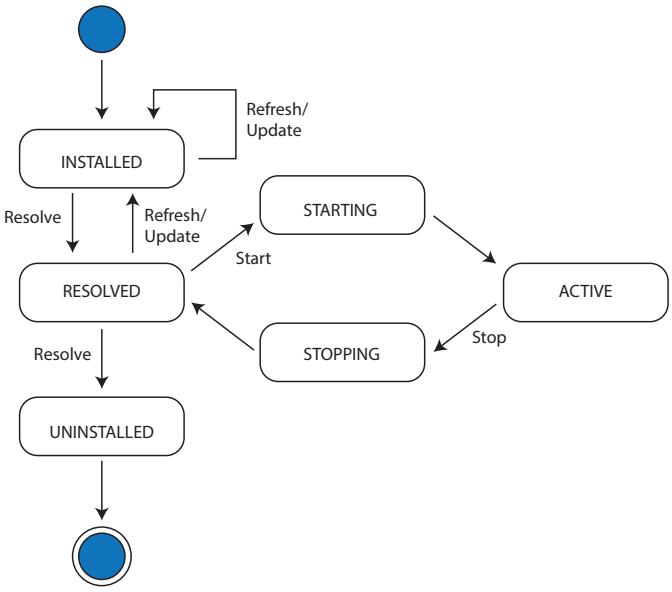


Figure 2.9: A life cycle of a bundle in OSGi

The OSGi specification defines the anatomy of a bundle. Figure 2.10 exhibits the anatomy of a bundle in an OSGi framework. We highlight that a bundle is just a JAR file that has a manifest file with a configuration about the bundle. In the manifest file, we specify the dependent bundles, the private packages, and so on of the bundle.

To install and uninstall a component at run-time, we can use the API provided by some implementation of the OSGi component framework specification. Many approaches use these APIs, provided by some framework that implemented the OSGi specification (i.e. Eclipse Equinox), to develop self-adaptive software systems (SASSs) using OSGi. Therefore, several studies typically use some OSGi framework as support to its approach.

Other OSGi bundles can use an OSGi bundle from two ways [116]: Sharing static code and exposing an OSGi service. The former is not good because in the manifest file belong to a bundle we statically have to declare the required bundles. On the other hand, the latter is better because it transparently simulates a Service-Oriented Architecture (SOA) [158].

A Bundle has to use the Services layer of the OSGi framework to use an OSGi service. The Services layer provides the registry of all the services and all the plumbing mechanisms

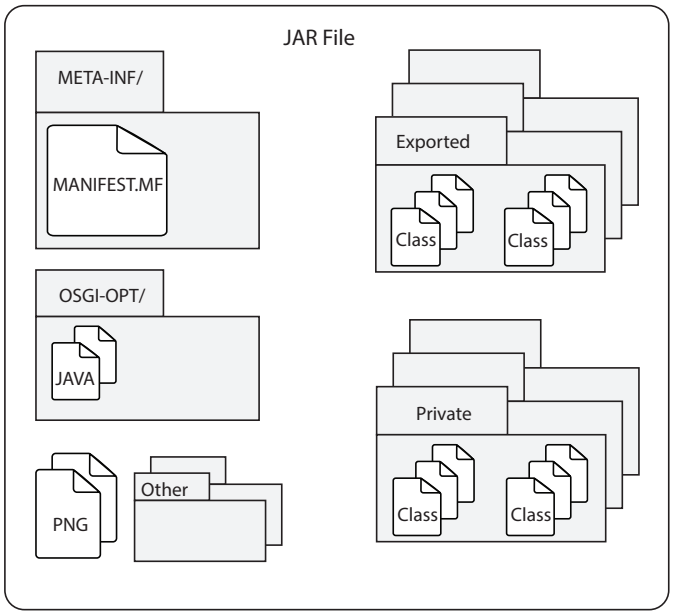


Figure 2.10: Anatomy of a bundle in OSGi

to wire these services [116]. Figure 2.11 depicts a Bundle B that requires an OSGi service, a Bundle A that implements an OSGi service, and a Bundle C that defines an OSGi service [101].

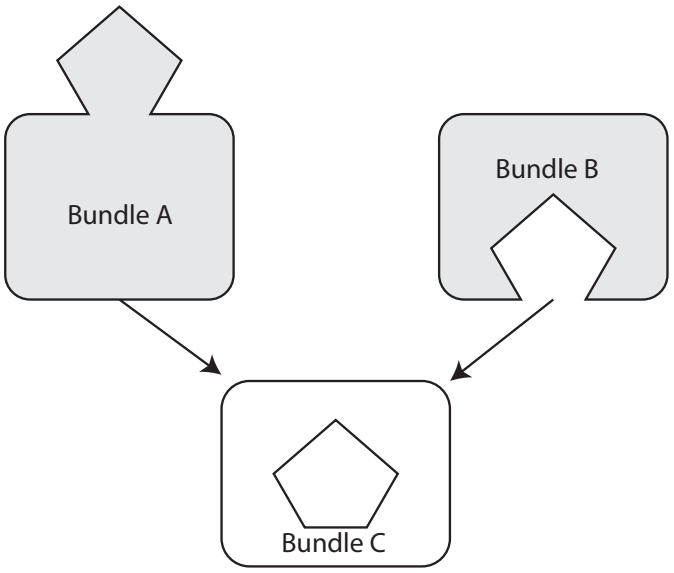


Figure 2.11: Exposing an OSGi service

2.4.2 COSMOS* implementation model

The COSMOS* component implementation model is an extension of the COSMOS component implementation model [53]. The COSMOS* implementation model defines a way of how to implement components and connectors using a set of design patterns in object-oriented programming languages such as Java and C++ [149]. Also, it allows reconfiguring components and connectors at design time. Figure 2.12 shows an architectural view of

a component that follows the COSMOS* component implementation model. Where the CompA component shown requires an interface called IB and provides two interfaces called IManager and IA.

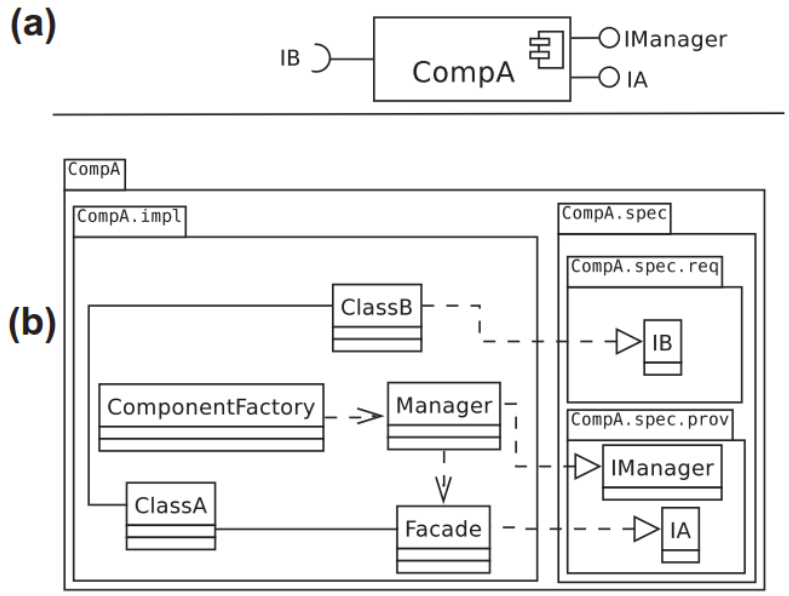


Figure 2.12: a) An architectural view of a component; b) A detailed design of a component that follows the COSMOS* component implementation model [149]

The COSMOS* implementation model defines that each component has a set of Facades classes (one for each provided interface), an IManager Interface, a Manager class, and a ComponentFactory class [53]. In each component, the ComponentFactory class has to create an instance of the Manager class. Moreover, each Facade class must implements, at least, a particular provided interface, and finally, each Manager class must manage the required and provided objects of the component at runtime.

When some object of the component needs to use a required object defined by a required interface, we can use the Manager object to obtain this object. We highlight that any class in the component can use this Manager object to acquire some stored instances that implement a required interface [53].

The COSMOS* implementation model defines that each connector must create an Adapter class, a Manager Class and a ComponentFactory class [53]. The Adapter class follows the Adapter design pattern [50], and then it adapts the provided and required interfaces between two components. The Manager class implements the IManager Interface, and then it has the same implementation as the IManager class of a component. And finally, the ComponentFactory class is used to obtain a instance of the Manager class.

2.5 Android architecture

The Android architecture is an open source software platform for mobile devices [38]. Android architecture is separated into five hierarchical categories [3]: applications, application framework, libraries, Android runtime, and Linux kernel. We highlight that

all Android applications typically use some components of the Android architecture to develop its functionalities. Figure 2.13 shows the hierarchical categories of the Android architecture with some of its principal subdivisions (or components) [32].

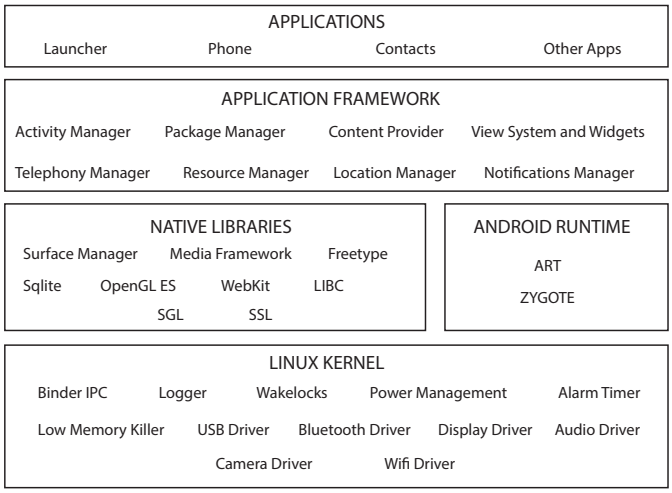


Figure 2.13: The Android architecture

We highlight that each Android application is packaged in a .apk archive for its future installation [141] in the platform. Our application mainly uses the components of the Application Framework category of the Android architecture (version 5.0.2). In Android 5.0, the Android Runtime (ART) is the official runtime for the Android applications [32]. In prior releases, it was the Dalvik Virtual Machine (VM).

2.6 Service-oriented architecture

A Service-Oriented Architecture is an architectural style that formally separates services from service consumers [150]. A service is some application logic that a system can provide [137] and a service consumer is a system that needs that application logic. SOA has three primary entities [104]: one or more service provider, one or more service consumers (requestor), and a service registry (broker). These entities interact through standard messaging protocols (e.g., HTTP and SOAP) that support the publishing, discovery and binding of services [66]. A service registry is a type of repository that allows companies to catalogue and reference the resources required to support the deployment and execution of services [106, 78].

Our application (See Chapter 4) uses REST services. A REST web service is an SOA based on the concept of "resource" (service) where the resources are stored on Uniform Resource Identifiers (URIs) [109]. A URI is a sequence of characters that identifies an abstract or physical resource [17].

2.7 Final remarks

The chapter showed the background required to understand the solution.

- Firstly, we showed several concepts relations to the dynamic software product line.
- Secondly, we showed the variability models. In particular, we presented the feature model.
- Thirdly, we associated the autonomic and self-adaptive system along with the self-* properties.
- Fourthly, we showed some technology to allow having architectural reconfigurations.
- Finally, we showed notions about SOA and the Android architecture to understand how we have implemented our Android application.

Chapter 3

The Cosmapek Infrastructure

Developers require an infrastructure to develop SASSs on Android devices. This infrastructure must have a Managing subsystem that reconfigures, reorganises, restructures or adapts the application (or Managed subsystem) itself at runtime. Therefore, the Managing subsystem has to manage at runtime and in some way the variability of the application. Additionally, this managing subsystem must have a tolerable performance and not must hinder the operations of the application. In this chapter, we present our Cosmapek infrastructure that seeks to fulfil the above requirements, along with its framework.

First, we show the foundations of our solution including the Dycosmos implementation model for runtime adaptations. The foundations of our solution are the adaptation space and graph, the mapping between features and architectural elements, and the dependency management. Second, we present the Cosmapek infrastructure with the architectural design of its framework and finally, we show a sequence of steps to create a self-adaptive software.

3.1 Foundations

3.1.1 Adaptation space and graph

To manage the variability at runtime, we need to know about the adaptation space and graph because the former is the variability itself and the latter is the structure that organises this variability.

We define an adaptation space as a conceptual space which represents all adaptations of the self-adaptive software system. When we use a variability model to model the variability of a system, we bound all adaptations of the modelled system. Therefore, we are bounding the adaptation space of the self-adaptive software.

We define an adaptation graph as a directed graph which shows all transitions between adaptions of a modelled system. In this graph, each node represents a set of adaptions, and each edge represents the possible transitions between nodes. To consider the dynamic or static properties (See 2.3.4) of a variability model, we have to define also two restrictions

- **Restriction 1** - We can not remove an element declared static of the variability model. We just can eliminate elements declared dynamic.

- **Restriction 2** - We have to remove all child elements (dynamic or static) of the selected element of the feature model.

The adaptation graph is generated by means a variability model. To build this graph, we have to use the algorithm 3.1. This algorithm uses the two above restrictions.

```
1  Input:
2      M: A variability model
3  Output:
4      An adaptation graph
5  begin
6     res := create a new adaptation graph
7     origin := create a new node(M)
8     res.add(origin)
9     for each node n of res
10      v := n.getVariabilityModel();
11      for each element e of v
12         // Restriction 1
13         if(e is dynamic){
14             v2 := create a copy of v
15             //remove this element and its children (Restriction 2)
16             v2.remove(e)
17             //update dynamic and static elements
18             v2.update()
19             if(res.exist(v2)){ //a node with this variability model
20                 n2 := res.get(v2) //get the node
21                 connect(n,n2,e) //n to n2 with edge e
22             }else{
23                 n2 := create a new node(v2)
24                 connect(n,n2,e)
25                 res.add(n2)
26             }
27         }
28     end
29     return res
30 end
```

Algorithm 3.1: To create an adaptation graph.

Four samples of adaptation space and graph are shown in the Figures 3.1, 3.2, 3.3 and 3.4. In each figure: a) shows the feature model; b) shows the adaptation space corresponding to the above feature model; and c) shows the adaptation graph corresponding to the same feature model. Besides, each node of this graph has a feature model which in turn has an adaptation space.

- **Sample 1** – In Figure 3.1 we show a parent feature with two mandatory features. Then, its adaptation space of this model has only one element. Applying the algorithm 3.1 to this feature model, we obtained an adaptation graph with one node (See Figure c) 3.1).

- **Sample 2** – In Figure 3.2 we show a parent feature with two optional features. Then, its adaptation space of this model has four elements. Applying the algorithm 3.1 to this feature model, we had an adaptation graph (See Figure c) 3.2) with four nodes.

- **Sample 3** – In Figure 3.3 we show a parent feature with two alternative features. Then, its adaptation space of this model has two elements. Applying the algorithm

3.1 to this feature model, we had an adaptation graph (See Figure c) 3.3) with three nodes.

- **Sample 4** – In Figure 3.4 we show a parent feature with two Or features. Then, its adaptation space of this model has three element. Applying the algorithm 3.1 to this feature model, we had an adaptation graph (See Figure c) 3.4) with three nodes.

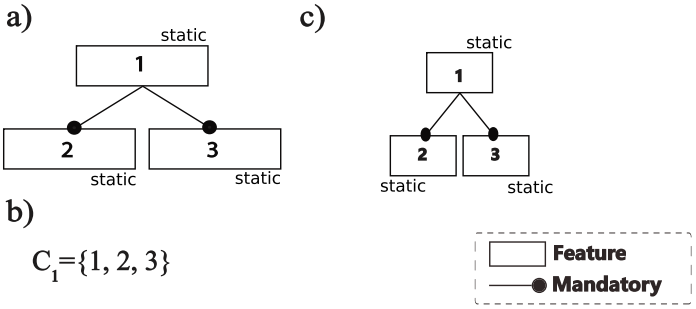


Figure 3.1: Adaptation space and graph of a Feature Model with Mandatory features

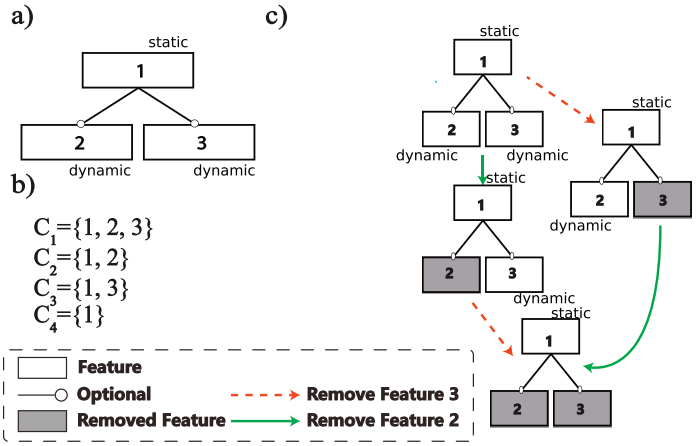


Figure 3.2: Adaptation space and graph of a Feature Model with Optional features

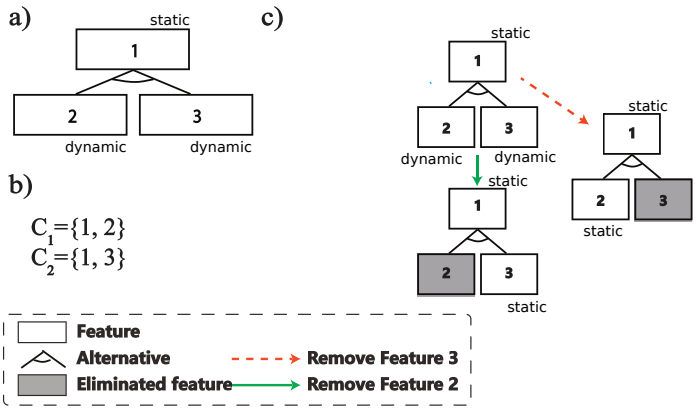


Figure 3.3: Adaptation space and graph of a Feature Model with Alternative features

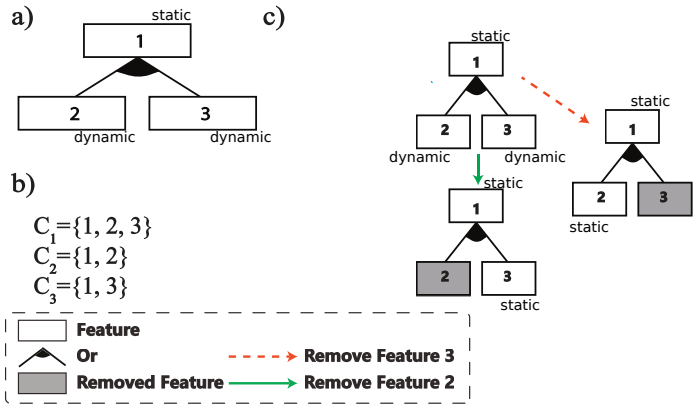


Figure 3.4: Adaptation space and graph of a Feature Model with Or features

3.1.2 Mapping between variability model elements and architectural elements

Because a variability model at runtime can have dynamic and static elements (See Section 2.2.3), we can also have dynamic and static architectural elements (AEs) at runtime. A dynamic architectural element can or cannot be taken into account at runtime; however, a static architectural element always has to be taken into account at runtime.

When a software is built using dynamic architectural elements, the upload and load of architectural elements at runtime generate a different software or system because it generates a different component architecture (See Section 2.4). Therefore, the use of a different architectural configuration (See Section 2.4) generates a different software or system.

To control the variability of a SASS using a variability model, we have to consider all the software that a SPL can generate as the adaptation states that a self-adaptive software (SAS) can have at run-time. This previous idea is not new, the dynamic software product lines (DSPLs) approach uses this idea as foundation of it (See Section 2.3.4).

Our infrastructure maps the variability model elements with the architectural elements of the software product line (SPL) to control at runtime the variability of the software using the variability model. In particular, it links the feature model elements with the architectural elements of the SPL architecture (See Section 2.1.3). Figure 3.5 shows an example of mapping between feature model elements and sets of architectural elements of an SPL architecture. We highlight that an item of a variability model can be linked to an empty set. In a feature model, this item represents an abstract feature.

In particular, Figure 3.5 shows a mapping of a feature model with five features and an architectural model with nine architectural elements. Therefore, a selection of features of the feature model will be mapped to a set of architectural elements. For instance, if we choose the F1, F3, and F5 features, in the architectural model we will have to choose and join the S1, S3, and S5 sets. As a result, our chosen features will be mapped to six architectural elements.

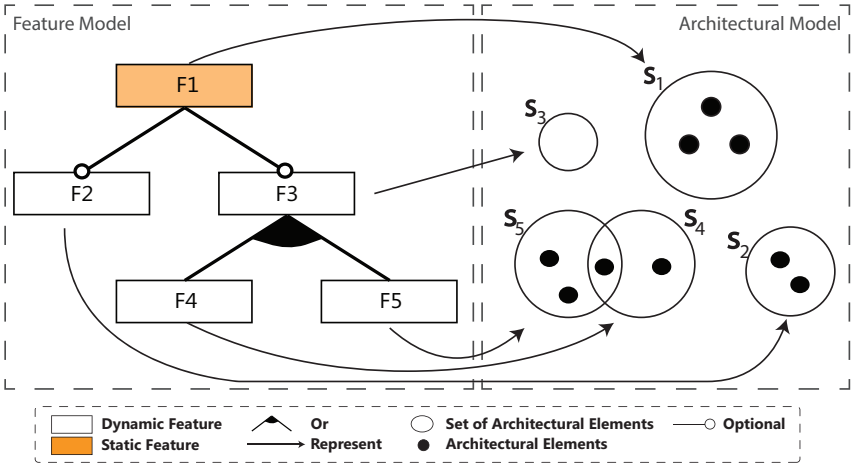


Figure 3.5: Mapping between features and architectural elements (EAs)

3.1.3 Dependency management

The loading or reconfiguration of architectural elements at runtime cannot be in a random manner because it can generate failures. Therefore, the loading or reconfiguration of these items has to be an order such that it does not produce failures. Fortunately, the architectural model shows the order of precedence between architectural elements.

To acquire this order of priority, we must follow two steps.

- Transform the architectural model to a directed graph.
- Apply the topology sorting algorithm to the directed graph.

We can apply the topology sorting algorithm only to directed graphs of architectural models without loops because these graphs do not have loops. Usually, an architectural model does not have loops, and in consequence, its representative directed graph will not have loops. Moreover, after employing the second step, we will get the order of precedence between architectural elements because each node with its weight represents some element of the architectural model.

Figure 3.6 shows an architecture model with its representative directed graph. Each component and connector of the architectural model has a representative node in the graph and each edge of the graph represents the dependency between architectural elements.

3.1.4 Dycosmos

Adaptations at runtime require of change at runtime between components and their connector. OSGI frameworks (See Section 2.4.1) allows obtaining these changes, loading and uploading bundles. However, managing OSGI bundles on Android platform is not an easy task, so we decide to create another way to control adaptations at runtime.

Dycosmos is one of our contributions. Dycosmos is a dynamic component implementation model which extends the Cosmos* component implementation model (See Section 2.4.2) such that the resulting components and connectors are reconfigurable at runtime.

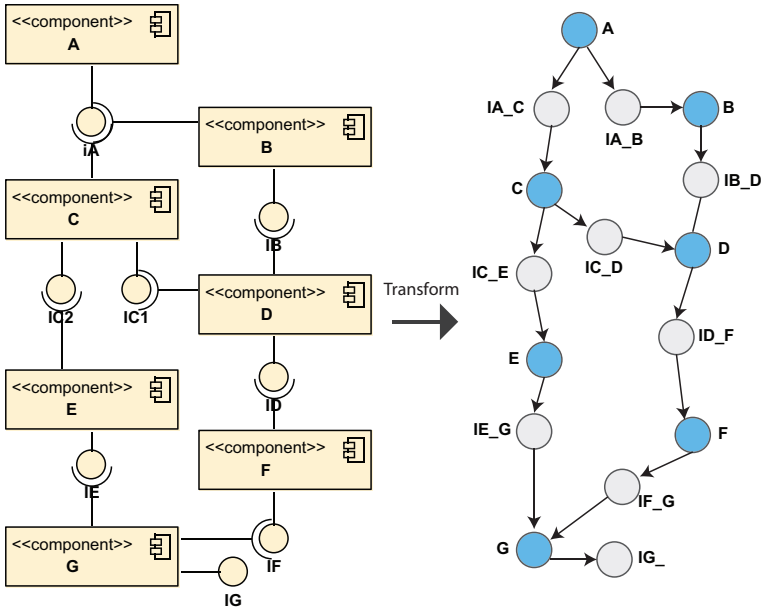


Figure 3.6: From an architectural model to a directed graph

Dycosmos added two main modifications to the previous implementation model to get reconfigurable connectors and components at runtime. The *createInstance* method of the *ComponentFactory* now creates a singleton object of the *Manager* class and all methods of the prior implementation model now are atomics. Furthermore, Dycosmos reorganises the structure of the Cosmos* implementation model.

Figure 3.7 shows a class diagram of a dynamic component, called *CompA*, implemented following the Dycosmos implementation model. This dynamic component shows a provided interface called IA and a required interface called IB. The Manager class implements four methods (setProvidedInterface, setRequiredInterface, getProvidedInterface and getRequiredInterface) what manages the variability at runtime of the components and connectors.

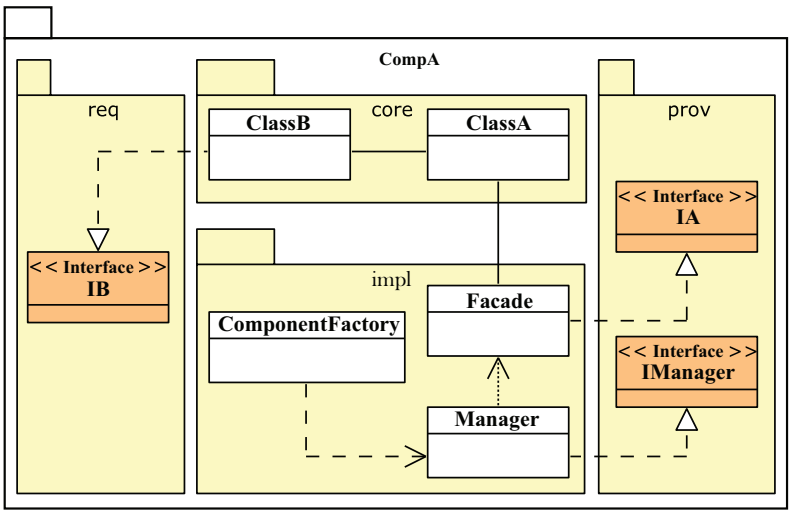


Figure 3.7: Class diagram of a dynamic component following Dycosmos

3.2 Cosmapek infrastructure

Figure 3.8 presents an overview of the Cosmapek Infrastructure, representing the managing subsystem and a deployed configuration of the managed subsystem. The managed subsystem, or application logic, is oblivious to the managing subsystem. The managing subsystem, or adaptation logic, is modularized following the MAPE-K loop (See Section 1.3). Moreover, each one of these modules has one or more implementation components. We highlight that each component or module shared a unique knowledge. In the following, we detail the managing subsystem, the managed subsystem, the knowledge base, and an adaptation scenario for tolerating a service failure.

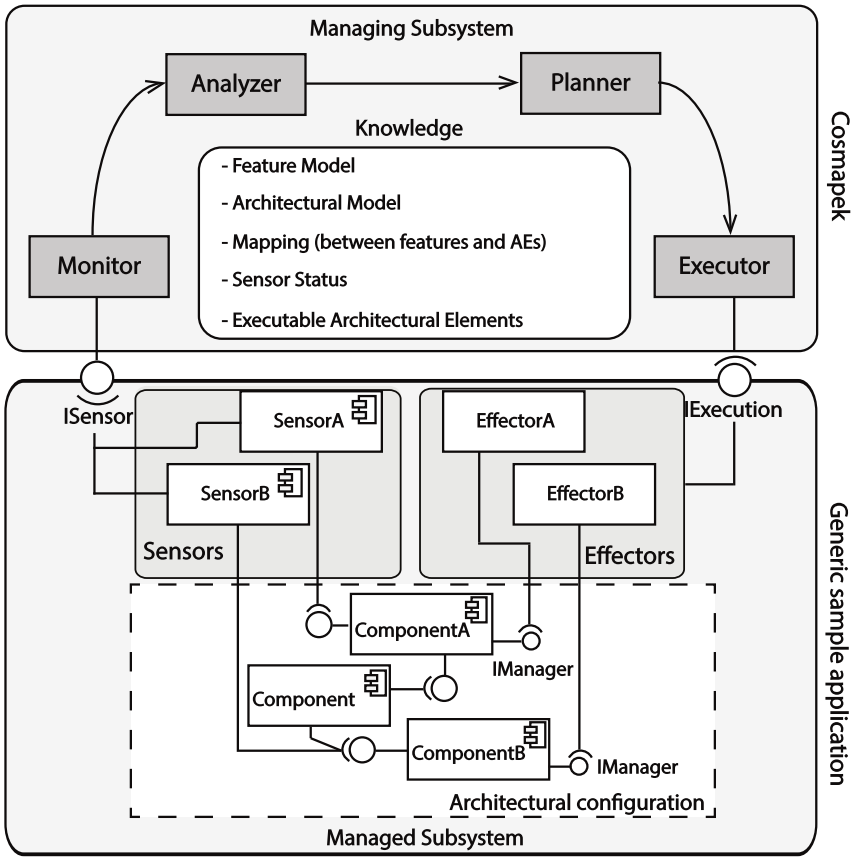


Figure 3.8: Overview of the Cosmapek infrastructure

3.2.1 The managing subsystem

The managing subsystem is independent of the application so that it can be reused across different domains. It incorporates the concepts of SASSs by providing an implementation of the MAPE-K loop. Figure 3.8 shows the main activities to perform an autonomic loop. In particular, the sensors observe the managed subsystem and send the collected information to the Monitor module. The Analyzer module, on the other hand, uses the data from the Monitor module to analyse the managed subsystem. The planner module generates architectural reconfiguration plans, selecting the most suitable architectural configuration, according to the knowledge base. Finally, the Executor module runs the effectors

to reconfigure the managed subsystem. As a result, the current deployed configuration changes.

3.2.2 The managed subsystem

The managed subsystem has to be a DSPL where a feature model represents the variability of the DSPL. In particular, we use the adaptation space and graph to limited the variability (See Section 3.1.1). The feature model contains a set of features, including static and dynamic features. Each feature linked some architectural element of the self-adaptive software. Static features are linked to static architectural elements and dynamic features are linked to dynamic architectural elements (See Section 3.1.2). The set of selected executable components and connectors that are running composes the current deployed architectural configuration of the managed subsystem.

From the viewpoint of Cosmapek, the managed subsystem is a client application. The DSPL must provide information about itself to allow managing subsystem of Cosmapek to control and to manipulate the current deployed architectural configuration of the managed subsystem.

Figure 3.8 shows a sample of a managed subsystem. In particular, this DSPL is developed based on services because of this separates services from service consumers. As a result, some components are service clients. The components connected to services will have sensors monitoring them. At runtime, the sensors will gather information about the availability of these services and will feed the managing system. Service unavailability or delay in response time are examples of failures detected by sensors.

When a service failure in the managed subsystem is detected, the managing subsystem can decide to change the deployed configuration by another valid architectural configuration using, for example, the benefits of dycosmos or OSGi. A valid architectural configuration is generated by the managing subsystem using data from the knowledge base.

3.2.3 The knowledge base

The models and artefacts that represent to the managed subsystem are stored in the knowledge base, as shown in Figure 3.8. The primary function of the knowledge is to feed the control loop in the managing subsystem. Cosmapek knowledge base is composed by:

- *Feature Model* – Represents the dynamic variability of the managed subsystem. It defines, regarding features, which configurations can be applied to the managed subsystem during its execution.

- *Architectural Model* – Represents the architectural model of the DSPL which the dynamic and static architectural elements. Also, it contains the priority order between the architectural elements to deploy (See Section 3.1.3).

- *Mapping* – Represents the mapping between features and architectural elements (See Section 3.1.2).

- *Sensor Status* – is a runtime hash table that registers the sensor status (activated or deactivated). In particular, the sensor status represents a flag that indicates whether there is a problem in the managed subsystem. This flag is used by managing subsystem to define if any action is needed.

- *Executable Architectural Elements* – are binary representations of components and connectors of the architectural model.

3.2.4 An adaptation scenario for tolerating service failures

When an application is deployed and executed, the adaptation loop from managing subsystem is activated. When a service used currently by the managed subsystem is not available, the sensor associated with service will detect and inform to Monitor module about this event. As a result, the Monitor module will save the status of the sensor in the knowledge. On the other hand, the Analyzer module will check the sensor status, using the knowledge, to analyse the deployed configuration of managed subsystem. The result of the analysis will indicate that the managed subsystem needs a reconfiguration because of one service used currently by the application is not available.

When the managed subsystem requires a reconfiguration, the Planner module is executed. It will prepare a new architectural configuration plan, according to the feature model and its mapping to architectural elements. In our scenario, an architectural configuration plan is composed by an architectural configuration, without the services that currently are not working. When the Planner module finishes the design of the plan, the Executor module begins to work. The Executor module uses this plan to execute (using the Java reflection API) the effectors associated with each element of the plan. Consequently, these effectors will reconfigure the managed subsystem and finally the application will work in a normal way.

3.3 Architectural design of the Cosmapek framework

Figure 3.9 presents a UML component diagram of the Cosmapek framework, that represents its architectural model. The model has three kinds of implementation components.

- Components that implement the functionality required by the MAPE-K loop – The *Monitor*, *Analyzer*, *Planner* and *Executor* components.

- Components that manage the data of knowledge base – The *Components*, *Features*, *Connectors* and *Variability* components.

- Components that support the operation of the managing subsystem – The *Reader* and *Controller* components.

The model refers only to the managing subsystem which has two extension points (*ISensor* and *IExecution* interfaces) for the development of the managed subsystem. The managing subsystem also provides and two control interfaces (*IControllerManager* and *IReaderManager*) to manages the Cosmapek.

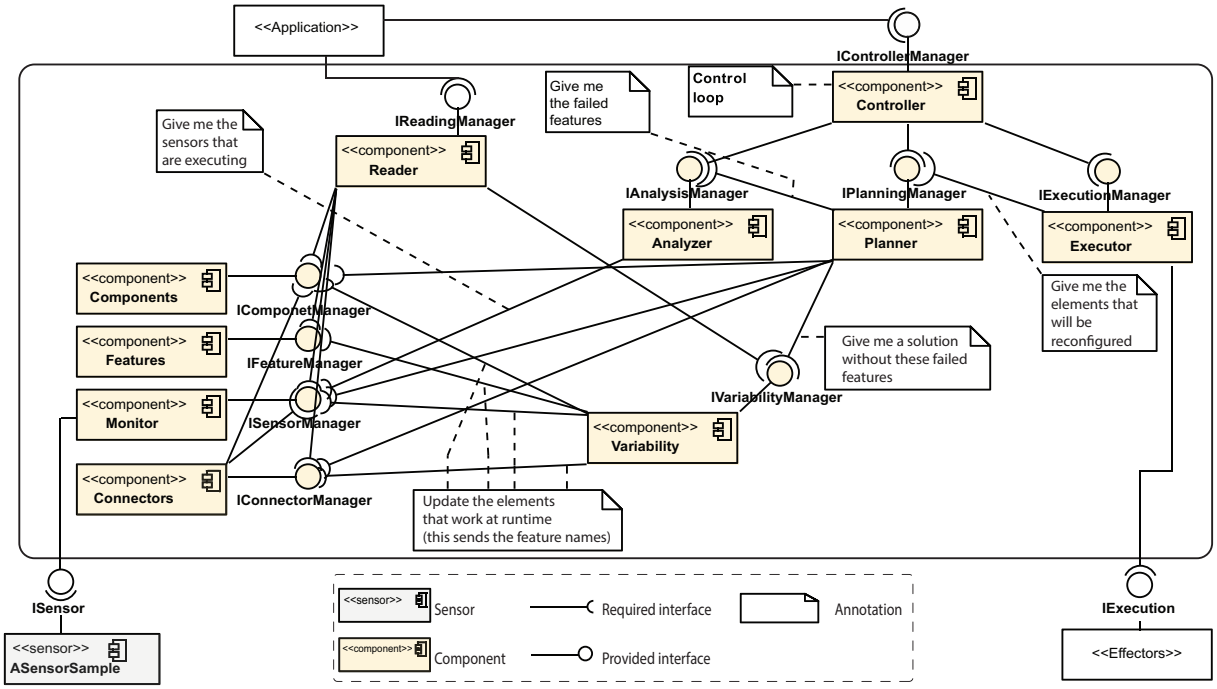


Figure 3.9: The architectural model of the Cosmapek infrastructure

- *IControllerManager Interface* – This interface is used by the App to specify the lapse of time that the infrastructure will use to analyse the register of sensors.
- *IReaderManager Interface* – This interface is used by the App to add the configuration files required by the Cosmapek.
- *ISensor Interface* – This interface is used by the sensors to update the record of the sensor status on the *Monitor* component.
- *IExecution Interface* – This interface is used to connect the effectors. Each effector implements this interface.

The Cosmapek framework has 8655 source code lines and it was implemented using ten customised components, following the Dycosmos implementation model (See Section 3.1.4).

3.3.1 Monitor

The *Monitor* component implements the Monitor module, as shown in Figure 3.9, it is responsible for monitoring the managed subsystem using the sensors. The *Monitor* component provides an interface called *ISensor* to the sensors sends information using this interface (See Figure 3.9). Sensors are a bridge between the managed and managing subsystem. They inform to the *Monitor* component some incident produced in the context or managed subsystem using the methods *activateSensor* and *desactivateSensor* of the *ISensor* interface.

3.3.2 Analyzer

The *Analyzer* component (Figure 3.9) implements the Analyzer module and it is responsible for analysing the state of the current deployed architectural configuration. The *Analyzer* component uses the information collected by the *Monitor* component to checks (at runtime) the state of the managed subsystem. An invalid feature indicates that it should be changed, and a valid feature symbolises that it can be used.

3.3.3 Planner

The *Planner* component (Figure 3.9) implements the Plan module and it is responsible for generating new architectural reconfiguration plans. This component performs a request to the *Analyzer* component to obtain the list of invalid feature and then makes a request to the *Variability* component to get a valid feature configuration at runtime which represents a valid product (See Section 3.1.1). The *Variability* component, which manages the variability binding at runtime, uses the adaptation graph and the SAT solver tool [91] to do the above task. In addition to this, the *Planner* component acquires the set of architectural elements mapped to the feature configuration using the assets from the knowledge base.

3.3.4 Executor

The *Executor* component, presented in Figure 3.9, implements the Executor module. This component runs the effectors of managed subsystem, following the architectural reconfiguration plan. The *Executor* component uses the Java reflection API to run the effectors. Cosmapek defines to have an effector associated with each architectural element of the architectural model.

Effectors are sequences of compiled commands, which at runtime execute reconfiguration actions on the managed subsystem without stopping the subsystem. As a result, the managed subsystem should provide, at runtime, reconfiguration mechanics of itself. We highlight that If some request in the managed subsystem is still being processed, the effector must wait until to complete this process. Finally, all effectors have to implement the *IExecution* interface.

3.3.5 Components

The *Components* component manages the updated register of components of the architectural model belong to adaptable software systems. This component allows that the framework knows which components are executing at runtime and also which features are associated with a particular component.

3.3.6 Connectors

The *Connectors* component manages the register of connectors of the architectural model that the adaptable software system holds. All stored information in the record of connec-

tors is supplied by the *Reader* and *Variability* components. Additionally, this component allows that the framework knows which connectors are executing at runtime.

3.3.7 Features

The *Features* component manages the register of the features belong to the adaptable software system. The information managed by the component is registered by the *Reader* and *Variability* components. Also, this component allows that the framework knows which features are executing at runtime.

3.3.8 Variability

The *Variability* component reads the configuration file that contains the variability of the system. In particular, this XML configuration file contains a feature model modelled using the featureIDE tool.

The *Variability* component loads the feature model to memory. Its main function is to provide a feature configuration without using a set of features. Moreover, the *Variability* component updates the *Features*, *Monitor*, *Components*, *Connectors* components with the result of the last query to the feature model.

3.3.9 Reader

The *Reader* component reads the configuration files required by the framework: the knowledge base and setting of the Cosmapek.

Firstly, the *Reader* component reads an XML file with the architectural model and the mapping (See Section 3.1.2). Secondly, the *Reader* component reads an XML file with the variability model (feature model) of the software system. In this case, the *Reader* component delegates the reading of it to the *Variability* component. Finally, the *Reader* component reads a path to the effectors. The framework will use this path to find effectors.

3.3.10 Controller

The *Controller* component generates part of the control loop of the infrastructure using the *Analyser*, *Planner* and *Executer* components. The period of monitoring is configured here.

3.4 How to use the Cosmapek

Figure 3.10 shows the three inputs that our managing subsystem requires.

- An XML file– which contains a feature model with the dynamic variability of managed subsystem.
- An XML file – that contains the architectural model and the mapping.
- A Path in the filesystem – where we can find the effectors of the system.

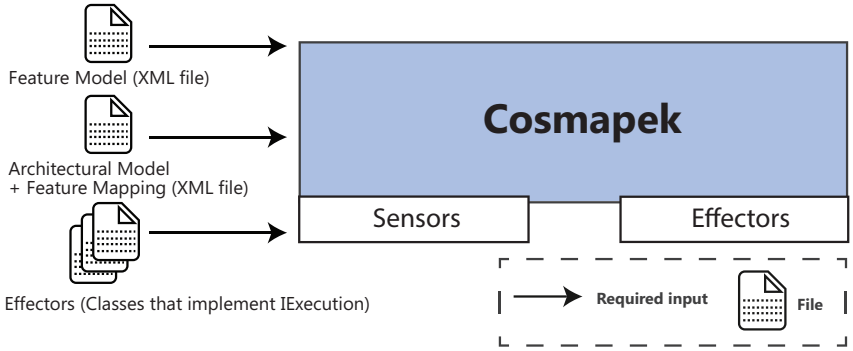


Figure 3.10: Required inputs of the Cosmapek infrastructure

In general, we can follow the next three steps to built a self-adaptive software.

- *Step 1* -– To design and create the managed subsystem as a DSPL, using a feature model to modelled the dynamic variability and a set of binary representations of components and connectors of the component architecture.
- *Step 2* -– To plan and create the effectors and sensors of the system. An effector for each element of the architectural model.
- *Step 3* -– To design and create the three inputs of the Cosmapek.
- *Step 4* -– To execute the Cosmapek in background.

3.4.1 Implementing a sensor

Sensors can be different software systems. However, to facilitate the develop of a self-adaptive software we described the sensors as components. Figure 3.11 shows how a sensor component can communicate something to Cosmapek framework. Additionally, these sensors must run on a different thread.

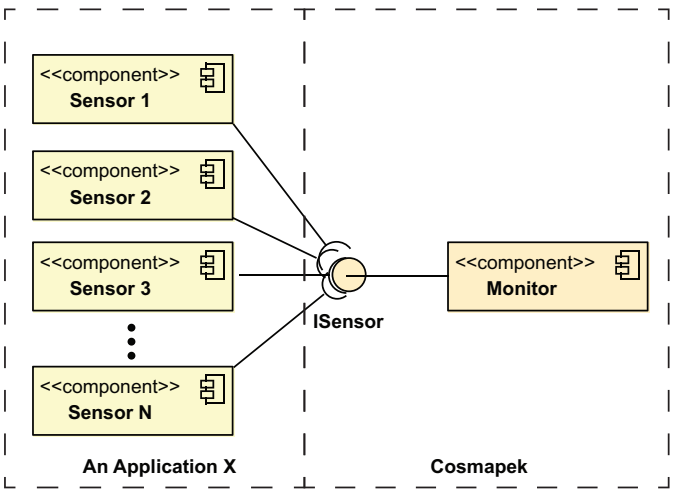


Figure 3.11: Sensors using the Cosmapek framework

3.4.2 Implementing an effector using Dycosmos

In general, we must create an effector using Dycosmos for each architectural element of the architectural model. Effectors that make reference to components should have commands that acquire the instance of the Manager object of the component (these commands also initialize the component). On the other hand, effectors that make reference to connectors (see figure 3.12) should have commands that initialize three architectural elements: the component that provided the required interface, the component that required the provided interface, and the connector of these two components. Besides, we must put commands that register the provided objects from first to the second component.

```
public class Conn_localizationB_company implements IExecution,
    IInterfaceTags {

private unicamp.buscame.localizationB.prov.IManager m_locaB;
private unicamp.buscame.localizationB.prov.ILocalizationManager i_locaB;
private unicamp.buscame.conn_localizationB_company.prov.IManager m_conn;
private unicamp.buscame.company.prov.IManager m_compa;

%@Override
public synchronized void execute() {
    //Acquire the instance
    m_locaB = unicamp.buscame.localizationB.impl.ComponentFactory.
        createInstance();
    //Get the provided object
    i_locaB = (unicamp.buscame.localizationB.prov.ILocalizationManager)
        m_locaB.getProvidedInterface(ILocalizationManagerTag);
    //Acquire the instance
    m_conn = unicamp.buscame.conn_localizationB_company.impl.
        ComponentFactory.createInstance();
    //Set the provided object
    m_conn.setRequiredInterface(ILocalizationManagerTag, i_locaB);
    //Acquire the instance
    m_compa = unicamp.buscame.company.impl.ComponentFactory.createInstance
        ();
    //Set the provided object
    m_compa.setRequiredInterface(ILocalizationManagerTag, m_conn.
        getProvidedInterface(ILocalizationManagerTag));
    }
}
```

Figure 3.12: An effector that makes reference to a connector

3.5 Final remarks

The chapter showed the foundation of our proposed solution and the Cosmapek infrastructure generated using the solution. Our adaptive deployment infrastructure uses techniques of SASS as a means to achieve the dynamic deployment.

The chapter also showed Dycosmos. Dycosmos is a new component implementation model that can be used to implement reconfigurable components and connectors at runtime.

Finally, the chapter showed the architectural design along with guidelines of how to use our solution.

Chapter 4

The Cosmapek Infrastructure Applied to Android

This chapter shows an Android application which is a self-adaptive application that follows our infrastructure and uses our framework.

We decided to implement our self-adaptive software system (SASS) on the Android platform instead of using other platforms (e.g. the desktop platform) because the number of Android applications in the world has grown exponentially [65]. In particular, the Android applications are written using the Java programming language, and many of these usually use the APIs that the Android software development kit (SDK) provides to the application developers [142].

First, the chapter shows the two general modules of the application: the client side and the server side. Specifically, we added the Cosmapek framework as a dependency of the client-side module without any modification. However, it had to be initialized and executed each time the application is initialized. Second, the chapter illustrates the sequence of steps that we took to add the self-adaptive behaviour to our application. Finally, we define and show the experiments and results.

4.1 The Buscame application

Our Android application allows that a user can search companies close to a location, using the latitude and longitude coordinates of Google Maps. Buscame has two general modules: the first one contains several REST API servers that are used by the application and the second one contains the Android application itself. The Cosmapek framework is located on the Android platform. We separated the application logic in REST servers to manipulate (turning on and off servers) the android application externally. Figure 4.1 shows the overview of the self-adaptive application. The application internally sends requests to REST API servers and receives responses from these servers.

4.1.1 The client side

This module contains our Android application (around 9320 source code line) together the Cosmapek infrastructure. Our application has seven main components, five alternate

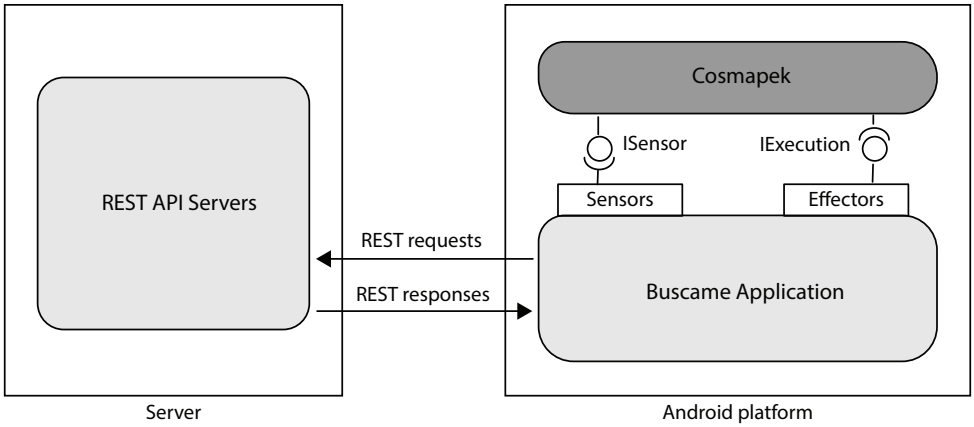


Figure 4.1: Modules of the Buscame application

components and eight components that work as sensors. Besides, all the components of this module were implemented following the DyCosmos implementation model.

- Main components: the Controller, UI, Company, Client, Product, Location and Configuration components
- Alternate components: The ProductB, ProductC, LocationB, LocationC and ConfigurationB components
- Sensor components: The ProductSensor, ProductBSensor, ProductCSensor, LocalizationSensor, LocalizationBSensor, LocalizationCSensor, ConfigurationSensor and ConfigurationBSensor components

The *Product*, *Configuration*, *Location* components and alternates components perform requests to the REST API serves to obtain the information that needed the components to implement the provided interfaces by these. In particular, each one of these components has associated a different REST API server. Figure 4.2 shows the architectural model associated to our Buscame. Also, we highlight that this architectural model is also the architectural model of a DSPL.

4.1.2 Components:

- **Controller component** is the component that manages all the functionality associated with the Android application. Additionally, this component uses the provided interfaces by the UI, Client and Company components to do its work.
- **UI component** is the component that manages the interfaces of users of the Android Application. Additionally, this component uses the provided interfaces by the Company and Client components to acquire the required information to generate and display the user interfaces.
- **Company component** is the component that manages everything related to a company. Additionally, this component uses the provided interfaces by Product, Localization and Configuration components to do its work.

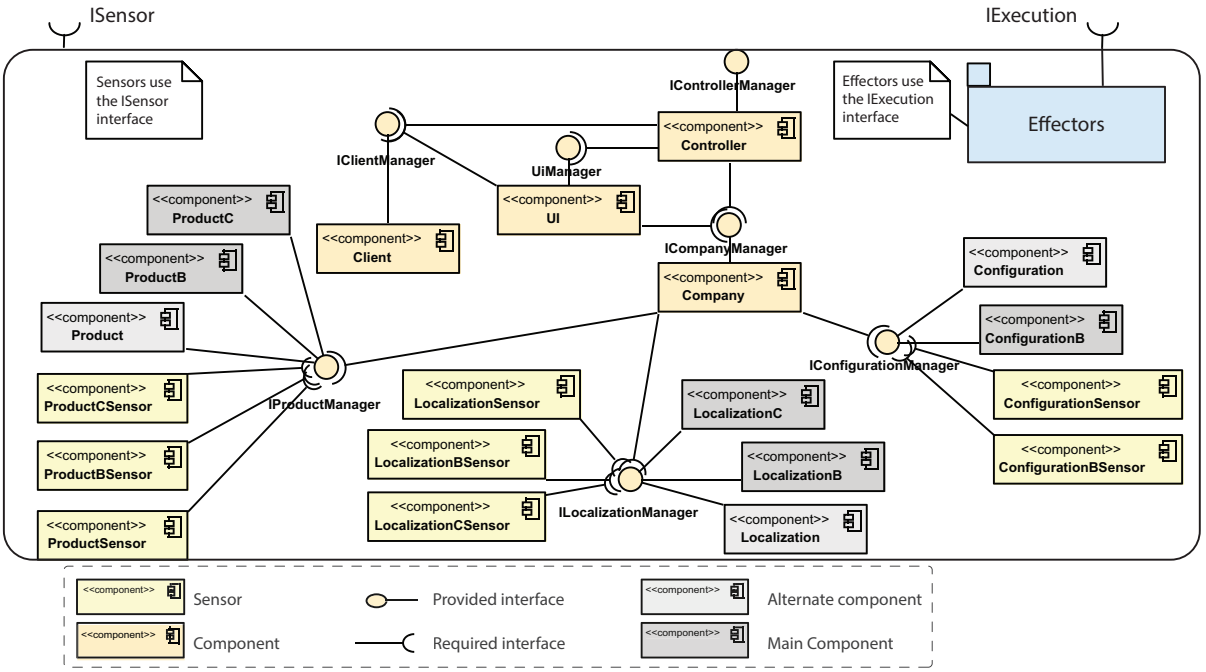


Figure 4.2: Architectural model associated to our Buscame

- **Client component** is the component that manages everything related to a client. Additionally, this component uses the sensors of location belong to the Android device to do its work.

- **Product component** is the component that manages everything related to a product. Additionally, this component sends REST requests to a REST API Server to obtain the required information about the products.

- **ProductB component** has the same functionality that the Product component. However, this element uses one different REST API Server to obtain the required information about the products.

- **ProductC component** has the same functionality that the Product component. However, this element uses one different REST API Server to obtain the required information about the products.

- **Localization component** is the component that manages the locations. Additionally, this component sends REST requests to a REST API Server to obtain the required information using a location.

- **LocalizationB component** has the same functionality as the Localization component. However, this element uses one different REST API Server to acquire the required information using a location.

- **LocalizationC component** has the same functionality as the Localization component. However, this element uses one different REST API Server to obtain the required information using a location.

- **Configuration component** is the component that manages everything related to a product configuration. Additionally, this component sends REST requests to a REST API Server to obtain the required information necessary for implementing its provided interfaces.

- **ConfigurationB component** performs the same task that the Configuration component performs, but using a different server.

- **ProductSensor component** is a sensor that monitors the Product component. Every certain period it uses the provided interfaces belong to Product component to query something.

  When the Product component throws some exception, the ProductSensor component catches this exception and register it in the framework. Specifically, the ProductSensor component records that the Product component at that moment is not working properly using the *ISensorUpdater* provided interface.

- **ProductBSensor component** is a sensor that monitors the ProductB component. And also it performs the same functionality as the ProductSensor Component.

- **ProductCSensor component** is a sensor that monitors the ProductC component. And also it performs the same functionality as the ProductSensor Component

- **LocalizationSensor component** is a sensor that monitors the Localization component. And also it performs the same functionality as the ProductSensor Component

- **LocalizationBSensor component** is a sensor that monitors the LocalizationB component And also it performs the same functionality as the ProductSensor Component

- **LocalizationCSensor component** is a sensor that monitors the LocalizationC component. And also it performs the same functionality as the ProductSensor Component

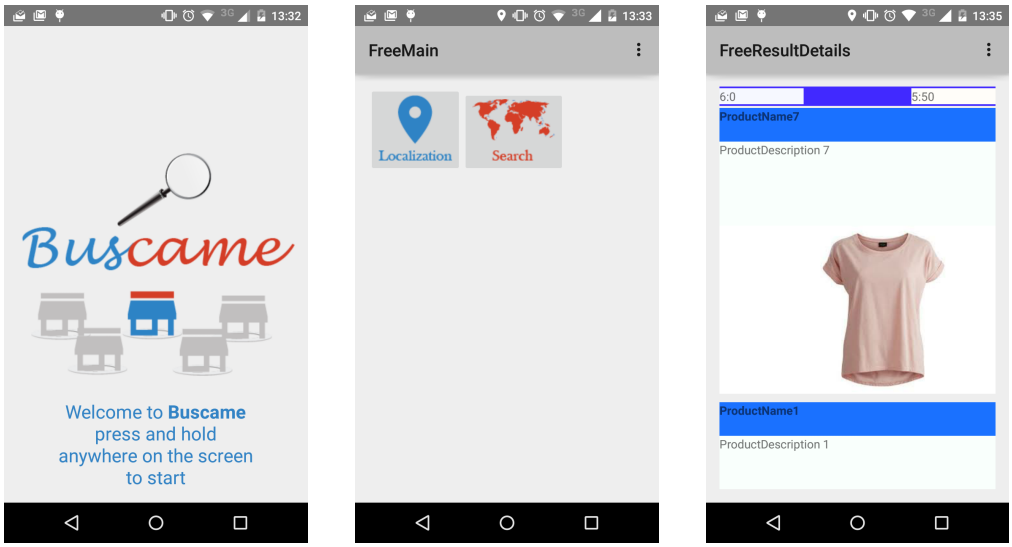
- **ConfigurationSensor component** is a sensor that monitors the Configuration component. And also it performs the same functionality as the ProductSensor Component

- **ConfigurationBSensor component** is a sensor that monitors the ConfigurationB component. Also, it performs the same functionality as the ProductSensor Component

4.1.3 User interface

Figure 4.3 shows the user interface (UI) of our Android application.

Our Android application has a menu (Menu UI) with two items: the Localization button and the Search button. When the user selects the localization button, our Android



(a) Initial UI (b) Menu UI (c) Products UI

(d) Location UI (e) Companies UI

Figure 4.3: User interfaces of Buscame

application shows the Location UI. And when the user selects the Search button, our application shows the Companies UI with the twenty companies closest to the current location of the user.

Moreover, when the user selects a location in the Location UI, the Android application shows the twenty companies closest to this new location (Companies UI). And then when the user selects a business in the Companies UI, the Android application shows the set of products that this company offers (Products UI).

4.1.4 Stages to adding self-adaptive behaviour

To add self-adaptive behaviour to some application (Android or Desktop), we mainly require the configurations files that our Cosmapek framework needs (See Section 3.4).

Developers of the SASS should design these setting files at design time.

- Firstly, we need to prepare the variability model using a feature model. We used the FeatureIDE plug-in to develop the feature model and then to generate the XML file associated with this model. Figure 4.4 shows the variability model of our Android application.

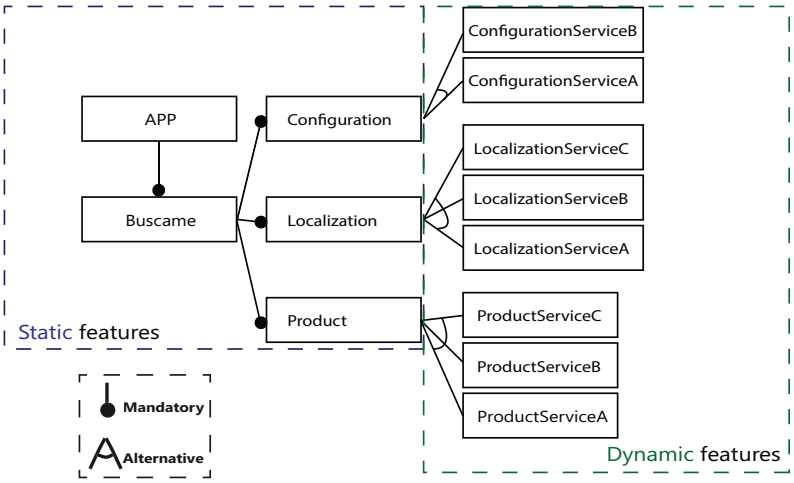


Figure 4.4: Feature model of the Buscame

- Secondly, we need to prepare the set of effectors and sensors of the system (See Sections 3.4.1 and 3.4.2). We created a set of classes (one for each element of the architectural model) to works as the effectors. These classes implement the *IExecution* interface of the framework and as a consequence, these implement the *execute* method of the *IExecution* interface.

- In third place, we need to create an XML file with the architectural model along with the configurations of the system.

- Finally, we need to add the prepared data to the Cosmapek framework and execute it on a different thread. To insert the input of the system, we use the IReadingManager interface of the framework and to run the framework we use the IControllerManager interface of the framework. From this moment forward Buscame is a self-adaptive software.

4.1.5 The server side

We have eight REST API servers with around 3202 source code lines. All our servers have the same functionality, the same stored information, and almost the same code. Each server provides a REST API on a different port and uses a separate database to store the information of the Buscame. Specifically, we use the OrientDB, NoSQL database management system, to manage the databases of the application.

Our servers were implemented using the Restlet framework. Being it a framework which maps the REST concepts to Java classes [96]. We emphasise that the Restlet framework can be used to implement any RESTful system [92].

We use a Quadtree structure to query companies closer to a location of Google Maps. In particular, we use a range query (or epsilon query). Also, all information, even the images, sent by the server to the client applications is stored in a JSON.

Moreover, when a request of "companies closer to a location" arrives at the server, the server (using the provided location) queries to the quadtree structure to obtain this information. Then the set of companies closest to the location obtained by the quadtree is ordered from minor to major. Finally, the server sends the first twenty companies.

The figure 4.5 shows the REST API servers of Buscame interacting with the components of our Android application.

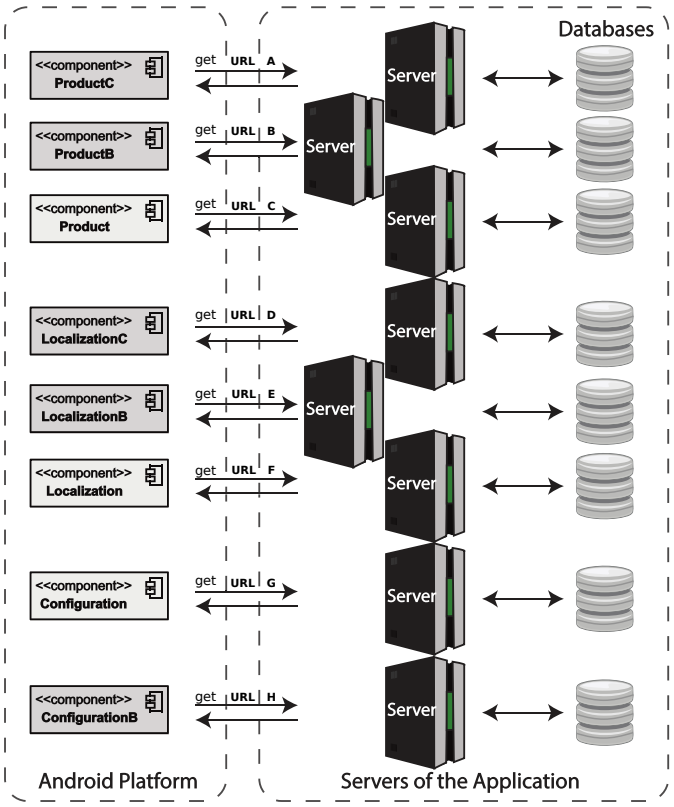


Figure 4.5: Servers of the Buscame application

4.2 Proof of concept of the Solution

Our target is to verify if our infrastructure really works on the Android platform. As a result, we designed and executed two exploratory experiments. In each experiment, the application and the Cosmapek infrastructure ran on Motorola Moto G (2nd Gen) with an Android platform (API 21 Lollipop). On the other hand, the servers and scripts ran on a Toshiba Laptop computer with a processor Intel(R) Core(TM) i7 CPU Q 720 @ 1.60GHz and a RAM memory of 4.00 GB.

4.2.1 Experiment 1 – Testing the Buscame application in exceptional scenarios

The target of the first experiment was to examine if the Buscame application using the Cosmapek infrastructure copes exceptional scenarios. To get our purpose, we designed two activities which aleatory applied during three hours.

- **Activity A** – In this activity, we turn off some currently used server, wait for the application reacts to this event and finally use some functionality.
- **Activity B** – In this activity, we only turn on some shutdown server and wait for the application responds to this event.

Besides, to know what is happening at runtime in the experiment, we added some position flags in the sensors and at the beginning and end of the Analyzer, Planner and Executor components of the framework.

When we run the experiment, we noticed using the position flags that the Buscame application changed several times its current architectural configuration in order to cope the incidents. Besides, we noticed that the sensors had a delay in detecting the disturbances. However, after detecting the incident, the Analyzer, Planner and Executor components of the managing subsystem had an acceptable performance. In Android applications, an acceptable performance means two to three seconds.

4.2.2 Experiment 2 – Testing the performance of the Cosmapek infrastructure

We executed a second experiment that consisted of measuring the performance of the Cosmapek. To simulate different environments in this experiment, we created scripts in Shell and Java that shut down and open up periodically, in a controlled manner, the servers of our self-adaptive application. Specifically, this experiment was of the type controlled because each simulated environment was aleatory built to have at least a solution. The time interval which our script generated a new environment was of the order of 5 minutes and 30 seconds.

In this experiment, we analysed and collected data (during four hours) of the infrastructure performance, in particular of the Analyser, Planner and Executor components in the Android platform. To obtain this information, we added position and time flags in these components.

Performance of the Analyser component

According to the data collected, the Analyser component analysed the system more of 3000 times because the control loop configuration of the Cosmapek was set to 5 seconds. Figure 4.6 shows exactly the points placed in the control loop where we took the initial and final time.

The obtained result shows that the time interval to perform an analysis of the system at runtime takes milliseconds, which is acceptable in some Android applications. Figure

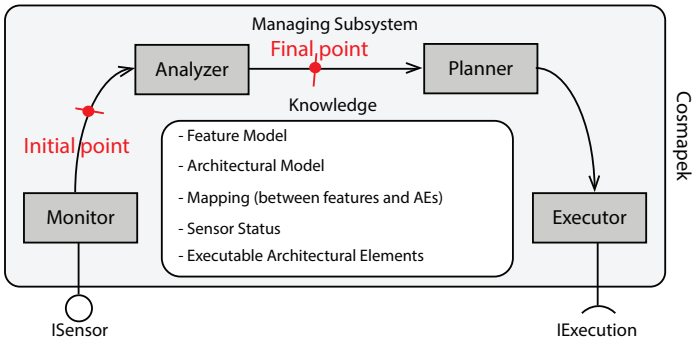


Figure 4.6: Initial and final points took in the experiment to measure the Analyzer component

4.7 shows this result. In this figure, the peaks are possibly generated by interference of the sensors. We highlight that this result did not include the time required for a sensor of the system to be activated or deactivated and the time needed for a sensor registers its current situation in the framework.

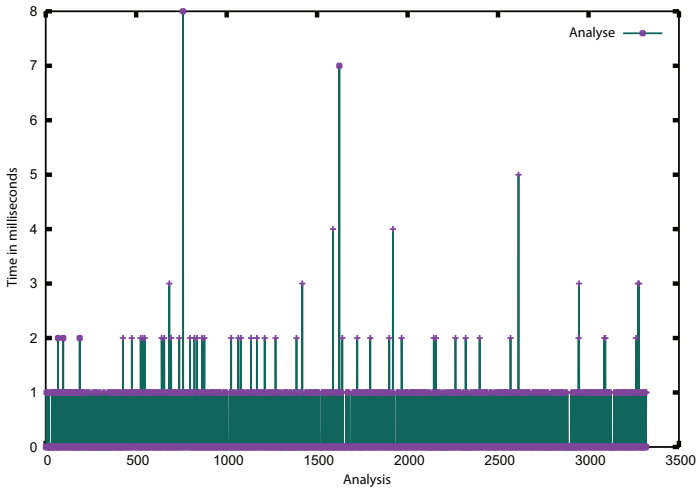


Figure 4.7: Time intervals obtained to perform an analysis of the system at runtime

Performance of the Planner component

According to the data collected, the Planner component executed 38 reconfiguration plans because the framework just allows creating a plan when a failure appear in some architectural element used at runtime. Figure 4.8 shows exactly the points placed in the control loop where we took the initial and final time.

The result obtained shows that the time interval to design a reconfiguration plan at runtime takes milliseconds which is acceptable in Android applications. The time varies depending on to the size and number of combinations of the FM. The FM of this App has 18 combinations. Figure 4.9 shows the results obtained of the experiment. We highlight that the queries to the FM to acquire a new valid features configuration of the SPL takes a considerable time because it was implemented using the Sat4j Java library.

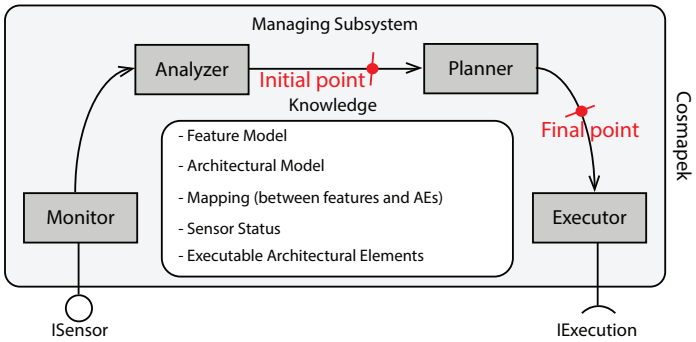


Figure 4.8: Initial and final points took in the experiment to measure the Planner component

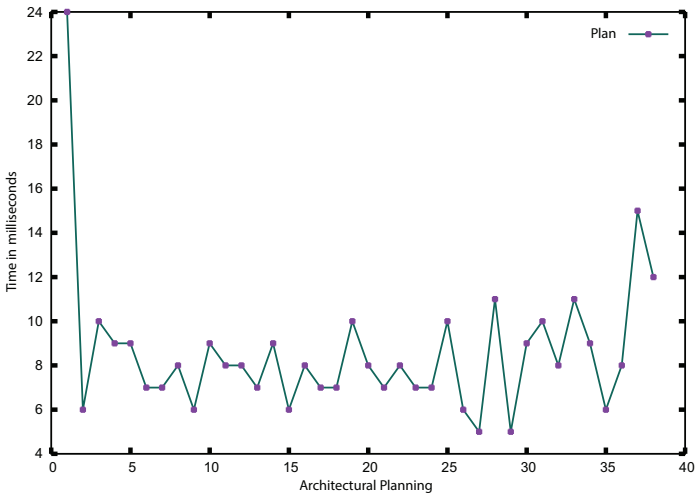


Figure 4.9: Time intervals obtained in designing of a plan at runtime

Performance of the Executor component

According to the data collected, the Executor component executed 38 architectural reconfiguration plans because the framework just allows running an architectural reconfiguration when some architectural plan was prepared before. Figure 4.10 shows exactly the points in the control loop where we took the initial and final time.

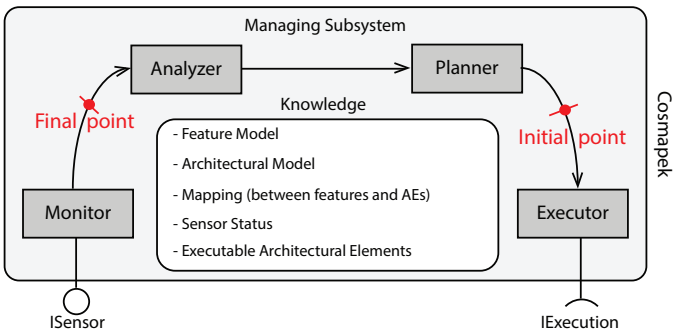


Figure 4.10: Initial and final points took in the experiment to measure the Executor component

The result obtained by applying the experiment indicated that the time interval to

perform an architectural reconfiguration at runtime, using the underlying properties of the Dycosmos implementation model in our application, takes milliseconds, which is acceptable in some Android applications. Figure 4.11 shows this result.

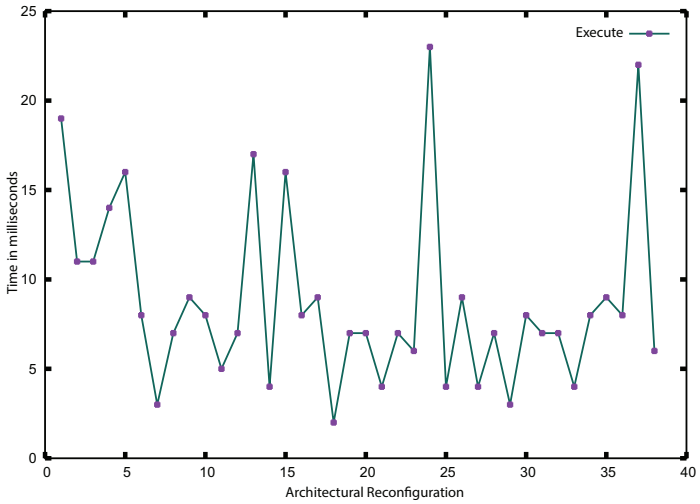


Figure 4.11: Time intervals obtained to perform an architectural reconfiguration at runtime

The above result indicates that the Dycosmos implementation model can replace the use of traditional OSGI frameworks.

Performance of the framework

The experiment collected 38 adaptations. However, some of these adaptations belong to the same scenario because our implemented sensors have a delay to detect a change in the environment and in some occasions these sensors are activated sequentially. In other words, n adaptation collected belong to the same scenario instead of different scenarios. Figure 4.12 shows exactly the points in the control loop where we took the initial and final time.

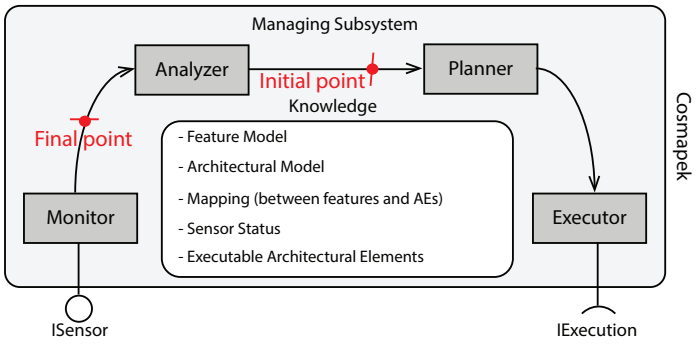


Figure 4.12: Initial and final points taken in the experiment to measure the Planner and Executor components

The result obtained by applying the experiment reveals that the time interval to perform an adaptation at runtime without regard to the time required to performance

the analysis of the system takes milliseconds. This time is acceptable in the Android applications. Figure 4.13 shows the result.

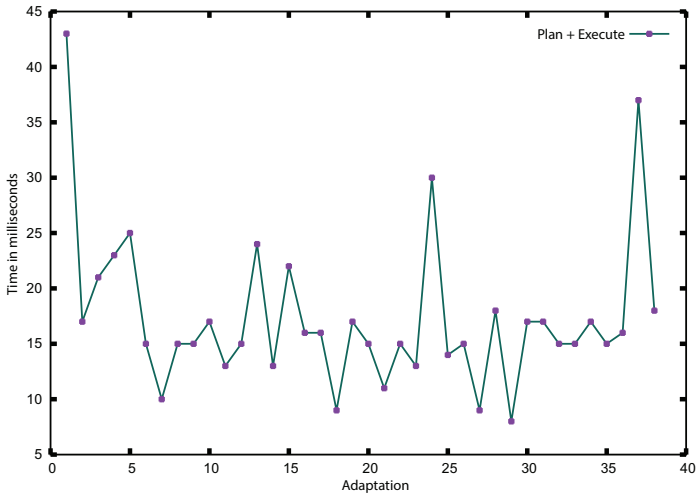


Figure 4.13: Time intervals obtained to perform an adaption at runtime

4.3 Final remarks

The chapter showed our self-adaptive software with their modules: client-side and server-side. In particular, the client-side module contains the Cosmapek framework, and the server-side module comprises the REST servers. The self-adaptive Android application tolerated the service unavailability by changing services at runtime.

Furthermore, the chapter showed two exploratory experiments to measure the performance and reliability of our solution.

- The first experiment showed that is possible achieve a self-adaptive behaviour on the Android platform which answers to the RQ1.
- The results of the second test showed that our solution has an acceptable performance, which solves to the RQ2. However, these results are depending on the application.

Finally, We highlight that our solution is one of the first solutions that works on Android platforms. Other works (See Section 1.4) only operating in desktop platforms.

Chapter 5

Conclusions and Future work

Our research generated a new infrastructure to the developing of self-adaptive software which facilitates the developing of this kind of systems on Android platforms. As a result, the research produced a new implementation model called Dycosmos.

Unlike other solutions (See Section 1.4), our solution was implemented to work on Android platforms. Besides, we provided a framework (implemented in Java) to support the use of our solution. According to our proof of concept, our solution has an acceptable performance in the Android world. Other works as the ArcMAPE and MoRE do not show an acceptable performance.

This chapter adds Dycosmos to the contributions of the study and besides it shows the conclusions and future work of our research.

5.1 Conclusions

- Managing architectural elements at runtime is the key to developing self-adaptive systems (SASs) in the software domain. This work has shown that is possible to construct a reconfigurable software (at runtime) using an object-oriented programming language (OOPL). Mainly, to implement the components and connectors of our reconfigurable software, we have used the Java programming language.

- Our new dynamic component implementation model, called Dycosmos, is a guide to code reconfigurable architectural elements at runtime. Our self-adaptive software was constructed following this new implementation model and had an acceptable performance when reconfiguring the system. Nonetheless, this new model added repetitive code to the software because of its design.

- A good software architecture is a key to facilitating the creation of software product lines and also the creation of dynamic software product lines. In this work, our application showed that a component-oriented architecture is ideal for creating DSPLs.

- Our proposed solution, presented in this work, places the control loop in the managing subsystem. The solution allowed creating a self-adaptive software on an Android

platform and favoured the creation of a new infrastructure to the developing of self-adaptive software.

- This work presented a new self-adaptive Android application called Buscame which coped failures that occur at runtime. Changing the use of services according to context, Buscame tolerated the service unavailability successfully. However, we had a poor performance detecting variations in the environment using our sensors (See Section 4.2.1).

5.2 New contributions

We add one new contribution to the mentioned in section 1.5.

Contribution - A new dynamic component implementation model called Dycosmos

As has already been mentioned in Section 3.1.4, Dycosmos is a extension of the Cosmos* component implementation model such that now the resulting components and connectors are reconfigurable at runtime.

5.3 Future work

Researchers can take some of the ideas presented here to continue this research.

- **Idea 1 - Replicate the study with a complete and new application**

  Our presented application is a small application. We need to build new applications, using a larger architecture and feature model, to better understand the behaviour of ou framework on these new self-adaptive systems.

- **Idea 2 - Add new functionality to our framework**

  Our framework can be updated to support distributed self-adaptive software systems. Thus, the new structure might be used to develop this kind of self-adaptive software. The work of Weyns et al. [162] can help to generate some new ideas to update our framework due to the fact it shows some patterns for a decentralised control.

- **Idea 3 - Create a new tool to create self-adaptive software systems using our infrastructure**

  We need to create a new tool to model software architectures and variability models. This new software must generate the architectural code of the architectural model using some dynamic component implementation model (e.g., Dycosmos) that allows built reconfigurable components and connectors at runtime. Also, this tool must generate the inputs required by the Cosmapek framework.

The tool has to be simple and without dependencies. Building an Eclipse plug-in using the Eclipse platform is not recommendable because this platform changes every year.

- **Idea 4 - Implement other variability models in our framework**

  For now, we are using the feature model to manage the variability of the self-adaptive software and, also, as the brain of this software. However, our implementation is not efficient and in a self-adaptive software with millions of features, we will need a new implementation or some new implementation of some variability model that can select at runtime the suitable features to some given context in a quick way.

- **Idea 5 - Apply new dynamic component implementation models**

  We have implemented our application using the Dycosmos component implementation model. However, other component implementation models can be applied. We propose to investigate or create new models similar to the Dycosmos implementation model to test our framework with these new models.

5.4 Final remarks

This chapter showed a new contribution of our list of contributions. Besides, it showed the conclusions and some new ideas to future works.

Bibliography

[1] Kevin MacG Adams. Understandability, Usability, Robustness and Survivability. In *Nonfunctional Requirements in Systems Analysis and Design*, number 28 in Topics in Safety, Risk, Reliability and Quality, pages 201–220. Springer International Publishing, 2015.

[2] Grant Allen. *Android Fragments*, pages 181–195. Apress, Berkeley, CA, 2015.

[3] Arisu An, H.-D.J. Jeong, Jiyoung Lim, and WooSeok Hyun. Design and Implementation of Location-Based SNS Smartphone Application for the Disabled Population. In *Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, pages 365–370. IEEE, 2012.

[4] Mikio Aoyama and Nozomi Kurono. An extended orthogonal variability model for metadata-driven multitenant cloud services. In *20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 339–346. IEEE, 2013.

[5] Mohsen Asadi, Samaneh Soltani, Dragan Gasevic, Marek Hatala, and Ebrahim Bagheri. Toward automated feature model configuration with optimizing nonfunctional requirements. *Information and Software Technology*, 56(9):1144–1165, 2014.

[6] R. Asadollahi, M. Salehie, and L. Tahvildari. StarMX: A framework for developing self-managing Java-based systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 58–67. IEEE, 2009.

[7] Colin Atkinson, Christian Bunse, Christian Peper, and Hans-Gerhard Gross. Component-Based Software Development for Embedded Systems – An Introduction. In *Component-Based Software Development for Embedded Systems*, number 3778 in Lecture Notes in Computer Science, pages 1–7. Springer Berlin Heidelberg, 2005.

[8] Pawel Bachara, Konrad Blachnicki, and Krzysztof Zielinski. Framework for application management with dynamic aspects J-EARS case study. *Information and Software Technology*, 52(1):67–78, 2010.

[9] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume ii: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.

[10] Randall Bachmeyer. *A Conceptual Graph Feature Model for Use in Developing Software Product-lines*. PhD thesis, University of Alabama in Huntsville, Huntsville, AL, USA, 2008.

[11] Wolfgang Banzhaf. Self-Organizing Systems. In *Encyclopedia of Physical Science and Technology (Third Edition)*, pages 589–598. Academic Press, 2003.

[12] Nilotpal Baruah, Lakshmi P Saikia, and K Hemachandran. System diagnosis and fault tolerance for distributed computing system: A review. *International Journal of Computer Science & Communication Networks*, 3(5):284, 2013.

[13] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[14] N. Bencomo, S. Hallsteinsen, and E. Santana de Almeida. A View of the Dynamic Software Product Line Landscape. *Computer*, 45(10):36–41, 2012.

[15] Nelly Bencomo, Robert B France, Betty HC Cheng, and Uwe Aßmann. *Models@run.time: foundations, applications, and roadmaps*, volume 8378. Springer, 2014.

[16] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, 2013.

[17] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform resource identifiers (uri): Generic syntax. 1998.

[18] B.S. Biswal and A. Mohapatra. Analogy of autonomic computing: An AIS (artificial immune systems) overview. In *International Conference on Computational Intelligence and Computing Research (ICCIC)*, pages 1–4. IEEE, 2014.

[19] Cristiana Bolchini, Marco Carminati, Antonio Miele, and Elisa Quintarelli. A framework to model self-adaptive computing systems. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 71–78. IEEE, 2013.

[20] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[21] Goetz Botterweck and Andreas Pleuss. Evolution of Software Product Lines. In *Evolving Software Systems*, pages 265–295. Springer Berlin Heidelberg, 2014.

[22] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[23] Alan W Brown and Kurt C Wallnau. The current state of cbse. *IEEE software*, 15(5):37, 1998.

[24] M. Butler. Android: Changing the Mobile Landscape. *IEEE Pervasive Computing*, 10(1):4–7, 2011.

[25] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[26] Dr Günter Böckle. Introduction to Software Product Line Engineering. In *Software Product Line Engineering*, pages 3–18. Springer Berlin Heidelberg, 2005.

[27] L. Cavallaro, E. Di Nitto, C.A. Furia, and M. Pradella. A Tile-Based Approach for Self-Assembling Service Compositions. In *2010 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 43–52. IEEE, 2010.

[28] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. *Computer*, 42(10):37–43, 2009.

[29] Shang-Wen Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the Rainbow self-adaptive system. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141. IEEE, 2009.

[30] Yu Cheng, A. Leon-Garcia, and I. Foster. Toward an Autonomic Service Management Framework: A Holistic Vision of SOA, AON, and Autonomic Computing. *IEEE Communications Magazine*, 46(5):138–146, 2008.

[31] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[32] Onur Cinar. *Android Quick APIs Reference*. Apress, 2015.

[33] Paul C. Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series. Addison-Wesley, Reading, MA, 1 edition, 2001.

[34] Philippa Conmy and Iain Bate. Assuring Safety for Component Based Software Engineering. In *15th International Symposium on High-Assurance Systems Engineering (HASE)*, pages 121–128. IEEE, 2014.

[35] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops (SPLC)*, pages 141–146. ACM, 2013.

[36] Chessman K. F. Corrêa. Towards Automatic Consistency Preservation for Model-driven Software Product Lines. In *Proceedings of the 15th International Software Product Line Conference (SPLC)*, pages 43:1–43:7. ACM, 2011.

[37] Javier Cámara, Rogério de Lemos, Nuno Laranjeiro, Rafael Ventura, and Marco Vieira. Robustness Evaluation of the Rainbow Framework for Self-adaptation. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC)*, pages 376–383. ACM, 2014.

[38] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *Information Security*, number 6531 in Lecture Notes in Computer Science, pages 346–360. Springer Berlin Heidelberg, 2010.

[39] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In *International Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 1–14. Springer, 2003.

[40] E.S. de Almeida, A. Alvaro, D. Lucredio, V.C. Garcia, and S.R. de Lemos Meira. RiSE project: towards a robust framework for software reuse. In *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI)*, pages 48–53, 2004.

[41] Kalyanmoy Deb and Hans-Georg Beyer. Self-Adaptive Genetic Algorithms with Simulated Binary Crossover. *Evolutionary Computation*, 9(2):197–221, 2001.

[42] Zuohua Ding, Yuan Zhou, and MengChu Zhou. Modeling Self-adaptive Software Systems with Learning Petri Nets. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 464–467. ACM, 2014.

[43] Francisco Assis Moreira do Nascimento, Marcio FS Oliveira, and Flávio Rech Wagner. A model-driven engineering framework for embedded systems design. *Innovations in Systems and Software Engineering*, 8(1):19–33, 2012.

[44] Ahmed Elfatatry. Dealing with Change: Components Versus Services. *Commun. ACM*, 50(8):35–39, 2007.

[45] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. FUSION: A Framework for Engineering Self-tuning Self-adaptive Software Systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE)*, pages 7–16. ACM, 2010.

[46] Emelie Engström and Per Runeson. Software product line testing – A systematic mapping study. *Information and Software Technology*, 53(1):2–13, 2011.

[47] B. Fitzgerald. Software Crisis 2.0. *Computer*, 45(4):89–91, 2012.

[48] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in Software Systems ;A Systematic Literature Review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2014.

[49] N. Gamez, L. Fuentes, and J.M. Troya. Creating Self-Adapting Mobile Systems with Dynamic Software Product Lines. *IEEE Software*, 32:105–112, 2015.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

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

[52] D. Garlan, Shang-Wen Cheng, An-Cheng Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[53] Leonel Aguilar Gayard. CosmosLoader: uma ferramenta de apoio ao gerenciamento de configuraçao baseado no modelo Cosmos. Master's thesis, University of Campinas, 2010.

[54] Walid Joseph Gédéon. *OSGi and Apache Felix 3.0 Beginner's Guide*. Packt Publishing Ltd, 2010.

[55] L. Gong. A software architecture for open service gateways. *IEEE Internet Computing*, 5(1):64–70, 2001.

[56] Jamie Goodyear and Johan Edstrom. *Instant OSGi Starter*. Packt Publishing, 2013.

[57] Ian Gorton, Yan Liu, and Nihar Trivedi. An Extensible, Lightweight Architecture for Adaptive J2ee Applications. In *Proceedings of the 6th International Workshop on Software Engineering and Middleware*, pages 47–54. ACM, 2006.

[58] Martin L Griss. Software reuse architecture, process, and organization for business success. In *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*, pages 86–89. IEEE, 1997.

[59] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and Model Checking Software Product Lines. In *Formal Methods for Open Object-Based Distributed Systems*, number 5051 in Lecture Notes in Computer Science, pages 113–131. Springer Berlin Heidelberg, 2008.

[60] Gabriela Guedes, Carla Silva, Monique Soares, and Jaelson Castro. Variability Management in Dynamic Software Product Lines: A Systematic Mapping. In *IX Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS)*, pages 90–99. IEEE, 2015.

[61] JP Gunderson and LF Gunderson. Intelligence= autonomy= capability. *Performance Metrics for Intelligent Systems, PERMIS*, 2004.

[62] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *Advances in Database Technology — EDBT '96*, number 1057 in Lecture Notes in Computer Science, pages 140–144. Springer Berlin Heidelberg, 1996.

[63] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008.

[64] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic Software Product Lines. In *Systems and Software Variability Management*, pages 253–260. Springer Berlin Heidelberg, 2013.

[65] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, number 7591 in Lecture Notes in Computer Science, pages 62–81. Springer Berlin Heidelberg, 2012.

[66] Andreas Helferich, Georg Herzwurm, Stefan Jesse, and Martin Mikusz. Software product lines, service-oriented architecture and frameworks: worlds apart or ideal partners? In *Trends in Enterprise Application Architecture*, pages 187–201. Springer, 2006.

[67] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing. In *Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 188–197. IEEE, 2013.

[68] Gabriel Hermosillo, Roberto Gomez, Lionel Seinturier, and Laurence Duchien. Using Aspect Programming to Secure Web Applications. *Journal of Software*, 2(6):53–63, 2007.

[69] Peter Herzum and Oliver Sims. *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley & Sons, Inc., 2000.

[70] M.G. Hinchey and R. Sterritt. Self-managing software. *Computer*, 39(2):107–109, 2006.

[71] Markus C. Huebscher and Julie A. McCann. A Survey of Autonomic Computing—Degrees, Models, and Applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, 2008.

[72] Nicklas Hult, Josefin Kadesjö, Björn Kadesjö, Christopher Gillberg, and Eva Billstedt. ADHD and the QbTest Diagnostic Validity of QbTest. *Journal of Attention Disorders*, page 1087054715595697, 2015.

[73] Alexandru F. Iosif-Lazăr, Ina Schaefer, and Andrzej Wasowski. A Core Language for Separate Variability Modeling. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, number 8802 in Lecture Notes in Computer Science, pages 257–272. Springer Berlin Heidelberg, 2014.

[74] JBoss AOP - JBoss Community. `http://jbossaop.jboss.org`. Accessed: 2016-02-02.

[75] Wootae Jeong. Sensors and Sensor Networks. In *Springer Handbook of Automation*, pages 333–348. Springer Berlin Heidelberg, 2009.

[76] Edson A. Oliveira Junior, Itana M. S. Gimenes, José C. Maldonado, Paulo C. Masiero, and Leonor Barroca. Systematic Evaluation of Software Product Line Architectures. *Journal of Universal Computer Science*, 19(1):25–52, 2013.

[77] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 611–614. IEEE Computer Society, 2009.

[78] L Frank Kenney, Daryl C Plummer, and Jess Thompson. Soa registries and policy enforcement bolster soa governance and consumption. *Gartner Research*, 2005.

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

[80] Minseong Kim, Sooyong Park, Vijayan Sugumaran, and Hwasil Yang. Managing requirements conflicts in software product lines: A goal and scenario based approach. *Data & Knowledge Engineering*, 61(3):417–432, 2007.

[81] Tariq M. King, Alain Ramirez, Peter J. Clarke, and Barbara Quinones-Morales. A Reusable Object-oriented Design to Support Self-testable Autonomic Software. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1664–1669. ACM, 2008.

[82] Kirk Knoernschild. *Java application architecture: modularity patterns with examples using OSGi*. Prentice Hall Press, 2012.

[83] M.M. Kokar, K. Baclawski, and Y.A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems and their Applications*, 14(3):37–45, 1999.

[84] B. Korherr and B. List. A UML 2 Profile for Variability Models and their Dependency to Business Processes (DEXA). In *18th International Workshop on Database and Expert Systems Applications*, pages 829–834. IEEE, 2007.

[85] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17, Part B:184–206, 2015.

[86] Chethana Kuloor and Armin Eberlein. Aspect-oriented requirements engineering for software product lines. In *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 98–107. IEEE Computer Society, 2003.

[87] Tiwari Umesh Kumar, Nautiyal Lata, Dimri Sushil Chandra, and Pal Ashish. Component Based Software Development in Distributed Systems. In *Mobile Communication and Power Engineering*, number 296 in Communications in Computer and Information Science, pages 56–61. Springer Berlin Heidelberg, 2013.

[88] Filip Křikava, Philippe Collet, and Robert B. France. Actor-based Runtime Model of Adaptable Feedback Control Loops. In *Proceedings of the 7th Workshop on Models@Run.Time*, pages 39–44. ACM, 2012.

[89] Robert Laddaga. Self Adaptive Software Problems and Projects. In *Second International IEEE Workshop on Software Evolvability*, pages 3–10, 2006.

[90] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O Automata for Interface and Product Line Theories. In *Programming Languages and Systems*, number 4421 in Lecture Notes in Computer Science, pages 64–79. Springer Berlin Heidelberg, 2007.

[91] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[92] Hong Jun Li. Research on restful web services in java. In *Applied Mechanics and Materials*, volume 135, pages 806–808. Trans Tech Publ, 2012.

[93] Xie Li and Wenjun Zhang. The design and implementation of home network system using OSGi compliant middleware. *IEEE Transactions on Consumer Electronics*, 50(2):528–534, 2004.

[94] Frank J Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering.* Springer Science & Business Media, 2007.

[95] Sihem Loukil, Slim Kallel, and Mohamed Jmaiel. Managing architectural reconfiguration at runtime. *International Journal of Web Portals (IJWP)*, 5(1):55–72, 2013.

[96] Jerome Louvel, Thierry Templier, and Thierry Boileau. *Restlet in Action: Developing RESTful Web APIs in Java.* Manning Publications Co., 2012.

[97] Frank D. Macias-Escriva, Rodolfo Haber, Raul del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18):7267–7279, 2013.

[98] S. Madria, V. Kumar, and R. Dalvi. Sensor Cloud: A Cloud of Virtual Sensors. *IEEE Software*, 31(2):70–77, 2014.

[99] Sara Mahdavi-Hezavehi, Matthias Galster, and Paris Avgeriou. Variability in quality attributes of service-based software systems: A systematic literature review. *Information and Software Technology*, 55(2):320–343, 2013.

[100] F. Maly and P. Kriz. Techniques for dynamic deployment of modules in context-aware Android applications. In *16th IEEE International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 107–111. IEEE, 2015.

[101] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating highly modular Java systems*. Addison-Wesley Professional, 2010.

[102] M Douglas McIlroy, JM Buxton, Peter Naur, and Brian Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98. SN, 1968.

[103] MD McIlroy. Software engineering: Report on a conference sponsored by the nato science committee. 1968.

[104] Dennis Medlow et al. Extending service-orientated architectures to the deployed land environment. *Journal of Battlefield Technology*, 13(1):27, 2010.

[105] Marc H Meyer and Alvin P Lehnerd. *The power of product platforms*. Simon and Schuster, 1997.

[106] A. Mintchev. Interoperability among Service Registry Implementations: Is UDDI Standard Enough? In *IEEE International Conference on Web Services (ICWS)*, pages 724–731. IEEE, 2008.

[107] Bardia Mohabbati, Marek Hatala, Dragan Gašević, Mohsen Asadi, and Marko Bošković. Development and Configuration of Service-oriented Systems Families. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1606–1613. ACM, 2011.

[108] Hausi A. Müller, H.M. Kienle, and U. Stege. Autonomic Computing Now You See It, Now You Don't — Design and Evolution of Autonomic Software Systems. *Software Engineering*, 5413 LNCS:32–54, 2009.

[109] Dzenana Muracevic and Haris Kurtagic. Geospatial soa using restful web services. In *Proceedings of the 31st International Conference on Information Technology Interfaces (ITI)*, pages 199–204. IEEE, 2009.

[110] Elisa Yumi Nakagawa, Pablo Oliveira Antonino, and Martin Becker. Reference Architecture and Product Line Architecture: A Subtle But Critical Difference. In *Software Architecture*, number 6903 in Lecture Notes in Computer Science, pages 207–211. Springer Berlin Heidelberg, 2011.

[111] S. Nakajima. Safe Substitution of Components in Self-Adaptive Web Applications. In *20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 388–395. IEEE, 2013.

[112] Shin Nakajima. An architecture of dynamically adaptive php-based web applications. In *18th Asia Pacific Software Engineering Conference (APSEC)*, pages 203–210. IEEE, 2011.

[113] A.S. Nascimento, C.M.F. Rubira, and F. Castor. ArCMAPE: A Software Product Line Infrastructure to Support Fault-Tolerant Composite Services. In *15th International Symposium on High-Assurance Systems Engineering (HASE)*, pages 41–48. IEEE, 2014.

[114] Peter Naur and Brian Randell. Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7-11 oct. 1968, brussels, scientific affairs division, nato. 1969.

[115] L.M. Northrop. SEI's software product line tenets. *IEEE Software*, 19(4):32–40, 2002.

[116] Jean-Baptiste Onofré. *Learning Karaf Cellar*. Packt Publishing Ltd, 2014.

[117] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and their Applications*, 14(3):54–62, 1999.

[118] Sebastian Oster, Andreas Wübbeke, Gregor Engels, and Andy Schürr. A survey of model-based software product lines testing. *Model-based Testing for Embedded System*, pages 339–381, 2011.

[119] Manish Parashar and Salim Hariri. Autonomic Computing: An Overview. In *Unconventional Programming Paradigms*, number 3566 in Lecture Notes in Computer Science, pages 257–269. Springer Berlin Heidelberg, 2005.

[120] D.L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[121] Gustavo G. Pascual, Roberto E. Lopez-Herrejon, Mónica Pinto, Lidia Fuentes, and Alexander Egyed. Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications. *Journal of Systems and Software*, 103:392–411, 2015.

[122] A. Petricic. Predictable dynamic deployment of components in embedded systems. In *33rd International Conference on Software Engineering (ICSE)*, pages 1128–1129. IEEE, 2011.

[123] Helmut Petritsch. Service-oriented architecture (soa) vs. component based architecture. *Vienna University of Technology, Vienna*, 2006.

[124] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[125] Jennifer Pérez, Nour Ali, Jose A. Carsí, and Isidro Ramos. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In *Component-Based Software Engineering*, number 4063 in Lecture Notes in Computer Science, pages 123–138. Springer Berlin Heidelberg, 2006.

[126] Awais Rashid, Jean-Claude Royer, and Andreas Rummler. *Aspect-oriented, model-driven software product lines: The AMPLE way*. Cambridge University Press, 2011.

[127] William N. Robinson and Yi Ding. A Survey of Customization Support in Agent-based Business Process Simulation Tools. *ACM Trans. Model. Comput. Simul.*, 20(3):14:1–14:29, 2010.

[128] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In *Software Engineering for Self-Adaptive Systems*, number 5525 in Lecture Notes in Computer Science, pages 164–182. Springer Berlin Heidelberg, 2009.

[129] Mazen Saleh and Hassan Gomaa. Separation of Concerns in Software Product Line Engineering. In *Proceedings of the Workshop on Modeling and Analysis of Concerns in Software*, pages 1–5. ACM, 2005.

[130] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, 2009.

[131] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, 2009.

[132] Mazeiar Salehie and Ladan Tahvildari. Towards a goal-driven approach to action selection in self-adaptive software. *Software: Practice and Experience*, 42(2):211–233, 2012.

[133] L. E. Sanchez, S. Moisan, and J. P. Rigault. Metrics on feature models to optimize configuration adaptation at run time. In *1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 39–44. IEEE, 2013.

[134] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *35th International Conference on Software Engineering (ICSE)*, pages 492–501. IEEE, 2013.

[135] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 119–126. ACM, 2011.

[136] P. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *14th IEEE International Conference on Requirements Engineering (RE)*. IEEE, September 2006.

[137] Christoph Schroth and Till Janner. Web 2.0 and soa: Converging concepts enabling the internet of services. *IT professional*, 9(3):36–41, 2007.

[138] Donald E Schwartz. Financial reporting of diversified companies: Legal implications. *Hastings LJ*, 20:119, 1968.

[139] Andy Schürr, Sebastian Oster, and Florian Markert. Model-Driven Software Product Line Testing: An Integrated Approach. In *SOFSEM 2010: Theory and Practice of Computer Science*, number 5901 in Lecture Notes in Computer Science, pages 112–131. Springer Berlin Heidelberg, 2010.

[140] Arun Sen. Metadata management: past, present and future. *Decision Support Systems*, 37(1):151–173, 2004.

[141] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-Powered Mobile Devices Using SELinux. *IEEE Security Privacy*, 8(3):36–44, 2010.

[142] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A Comprehensive Security Assessment. *IEEE Security Privacy*, 8(2):35–44, 2010.

[143] Liwei Shen, Xin Peng, Jindu Liu, and Wenyun Zhao. Towards Feature-oriented Variability Reconfiguration in Dynamic Software Product Lines. *Top Product. through Softw. Reuse*, 6727:52–68, 2011.

[144] Liwei Shen, Xin Peng, Jindu Liu, and Wenyun Zhao. Towards Feature-Oriented Variability Reconfiguration in Dynamic Software Product Lines. In Klaus Schmid, editor, *Top Productivity through Software Reuse*, number 6727 in Lecture Notes in Computer Science, pages 52–68. Springer Berlin Heidelberg, 2011.

[145] Pooja Soni and Nisha Ratti. Analysis of component composition approaches. *International Journal of Computer Science and Communication Engineering*, 2(1), 2013.

[146] Jacob Swanson, Myra B. Cohen, Matthew B. Dwyer, Brady J. Garvin, and Justin Firestone. Beyond the Rainbow: Self-adaptive Failure Avoidance in Configurable Systems. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE)*, pages 377–388. ACM, 2014.

[147] Richard N. Taylor. The Role of Architectural Styles in Successful Software Ecosystems. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 2–4. ACM, 2013.

[148] S. Thiel and A. Hein. Modelling and using product line variability in automotive systems. *IEEE Software*, 19(4):66–72, 2002.

[149] Leonardo P. Tizzei, Marcelo Dias, Cecília M. F. Rubira, Alessandro Garcia, and Jaejoon Lee. Components meet aspects: Assessing design stability of a software product line. *Information and Software Technology*, 53(2):121–136, 2011.

[150] Jihed Touzi, Fréderick Benaben, Hervé Pingaud, and Jean Pierre Lorré. A model-driven approach for collaborative service-oriented architecture design. *International Journal of Production Economics*, 121(1):5–20, 2009.

[151] I. van de Weerd, S. Brinkkemper, R. Nieuwenhuis, J. Versendaal, and L. Bijlsma. Towards a Reference Framework for Software Product Management. In *14th IEEE International Conference on Requirements Engineering (RE)*, pages 319–322. IEEE, 2006.

[152] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 45–54. IEEE Computer Society, 2001.

[153] Jilles van Gurp and Juha Erik Savolainen. Service Grid Variability Realization. In *10th International Conference on Software Product Line (SPLC)*, pages 85–94. IEEE, 2006.

[154] Martin Verlage and Thomas Kiesgen. Five Years of Product Line Engineering in a Small Company. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 534–543. ACM, 2005.

[155] Thomas Vogel and Holger Giese. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Trans. Auton. Adapt. Syst.*, 8(4):18:1–18:33, 2014.

[156] G.M. Waku, E.R. Bollis, C.M.F. Rubira, and R. Da S.Torres. A Robust Software Product Line Architecture for Data Collection in Android Platform. In *IX Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS)*, pages 31–39. IEEE, 2015.

[157] G.M. Waku, C.M.F. Rubira, and L.P. Tizzei. A Case Study Using AOP and Components to Build Software Product Lines in Android Platform. In *41st Euromicro Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA)*, pages 418–421. IEEE, 2015.

[158] Craig Walls. *Modular java: creating flexible applications with OSGi and Spring*. CreateSpace Independent Publishing Platform, 2014.

[159] Steven Wartik and Ted Davis. A phased reuse adoption model. *Journal of Systems and Software*, 46(1):13–23, 1999.

[160] Bernhard Westfechtel and Reidar Conradi. Software Architecture and Software Configuration Management. In *Software Configuration Management*, number 2649 in Lecture Notes in Computer Science, pages 24–39. Springer Berlin Heidelberg, 2003.

[161] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. On patterns for decentralized control in self-adaptive systems. *Lecture Notes in Computer Science*, 7475 LNCS:76–107, 2013.

[162] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II*, Lecture Notes in Computer Science, pages 76–107. Springer Berlin Heidelberg, 2013.

[163] Kostyantyn Yermashov. *Software composition with templates.* PhD thesis, De Montfort University, 2008.

[164] Eric Yuan, Sam Malek, Bradley Schmerl, David Garlan, and Jeff Gennari. Architecture-based Self-protecting Software Systems. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA)*, pages 33–42. ACM, 2013.

[165] Xiaorui Zhang and Birger Møller-Pedersen. *Towards correct product derivation in model-driven product lines.* Springer, 2012.