Continuous deployment of pervasive applications in dynamic environments

Ozan Necati Günalp

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Necati Ozan GÜNALP

Thèse dirigée par **Philippe LALANDA**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Déploiement continu des applications pervasives en milieux dynamiques

Thèse soutenue publiquement le **13 Novembre 2014**,
devant le jury composé de :

Mme Frédérique LAFOREST
Professeur à Université de Saint Etienne, Présidente
Mr Christian BECKER
Professor at Universität Mannheim, Rapporteur
Mr Frédéric WEIS
Professeur à l'Université de Rennes, Rapporteur
Mr Philippe LALANDA
Professeur à Université Joseph Fourier, Directeur de thèse
Mr Clément ESCOFFIER
Ingénieure de Recherche, Université Joseph Fourier, Co-Encadrant de thèse
Mr Vincent LESTIDEAU
Maitre de Conférences, Université Joseph Fourier, Co-Encadrant de thèse
Mr Levent GÜRGEN
Ingénieure de Recherche, CEA-LETI, Co-Encadrant de thèse

From too much love of living,
From hope and fear set free,
We thank with brief thanksgiving
Whatever gods may be
That no life lives for ever;
That dead men rise up never;
That even the weariest river
Winds somewhere safe to sea.

— Charles Swineburne,
The Garden of Proserpine

To my dear family and friends.

Continuous deployment of pervasive applications in dynamic environments

Abstract

Driven by the emergence of new computing environments, dynamically evolving software systems makes it impossible for developers to deploy software with human-centric processes. Instead, there is an increasing need for automation tools that continuously deploy software into execution, in order to push updates or adapt existing software regarding contextual and business changes. Existing solutions fall short on providing fault-tolerant, reproducible deployments that would scale on heterogeneous environments.

This thesis focuses on enabling continuous deployment solutions for dynamic environments, such as Pervasive Computing environments. It adopts an approach based on a transactional, idempotent process for coordinating deployment actions. This thesis proposes a set of deployment tools, including a deployment manager capable of conducting deployments and continuously adapting applications according to the changes in the current state of the target platform. The implementation of these tools, Rondo, also allows developers and administrators to code application deployments thanks to a deployment descriptor DSL.

Using the implementation of Rondo, the propositions of this thesis are validated in several industrial and academic projects by provisioning frameworks as well as installing applications and continuous reconfigurations.

Résumé

L'émergence des nouveaux types d'environnements informatiques amplifie le besoin pour des systèmes logiciels d'être capables d'évoluer dynamiquement. Cependant, ces systèmes rendent très difficile le déploiement de logiciels en utilisant des processus humains. Il y a donc un besoin croissant d'outils d'automatisation qui permettent de déployer et reconfigurer des systèmes logiciels sans en interrompre l'exécution. Le processus de déploiement continu et automatisé permet de mettre à jour ou d'adapter un logiciel en exécution en fonction des changements contextuels et des exigences opérationnelles. Les solutions existantes ne permettent pas des déploiements reproductibles et tolérant aux pannes dans des environnements fluctuants, et donc requérant une adaptation continue.

Cette thèse se concentre en particulier sur des solutions de déploiement continu pour les plates-formes d'exécution dynamiques, tels que celle utilisé dans les environnements ubiquitaires. Elle adopte une approche basée sur un processus transactionnel et idempotent pour coordonner les actions de déploiement. La thèse propose, également, un ensemble d'outils, y compris un gestionnaire de déploiement capable de mener des déploiements discret, mais également d'adapter les applications continuellement en fonction des changements contextuels. La mise en œuvre de ces outils, permet notamment aux développeurs et aux administrateurs de développer des déploiements d'applications grâce à un langage spécifique suivant les principes de l'‘infrastructure-as-code.

En utilisant l'implantation de Rondo, les propositions de cette thèse sont validées dans plusieurs projets industriels et académiques à la fois pour l'administration de plates-formes ubiquitaires ainsi que pour l'installation d'applications et leurs reconfigurations continues.

Contents

Foreword

I present with this manuscript the work I conducted during four years of doctoral studies. The first three years of this period was a result of the collaboration between the research team Adèle of Laboratoire Informatique de Grenoble (LIG) and the Lialp team of Commissariat à l'énergie atomique et aux énergies alternatives (CEA). The work of my final year, which concluded my studies, was carried out entirely in Adèle.

These four years have been a great journey, not short of many ups and downs. I would like to take my time here to thank some of the people who made this possible and helped me to end up with this work.

I present my gratitude to Christian Becker and Frédéric Weis for accepting to review this thesis and Frédérique Laforest for examining my work.

First of all, I would like to thank my supervisor, Philippe Lalanda, for giving me the opportunity to be a part of the Adèle team, providing me the guidance and support to perform research. I would also like to thank Levent Gürgen, who welcomed me in the CEA and gave me the possibility to contribute in numerous projects. Of course this work would not be possible without Vincent Lestideau and Clément Escoffier, who despite their limited availability, were always keen to guide me with precious remarks and encouragements.

I wish to thank all members of Adèle and Lialp teams, with whom I shared 5 years plenty of good moments. All of them helped me in this work, through fruitful discussions and advice. Stéphanie, German, Denis, Jonathan, Johann, Eric, Walter, Diana, Pierre, Torito, Gabriel, Bassem, Morgan, Etienne, Yoann, Jander, Colin, Thibaud, Suzanne, Yeter, Safietou, Mathieu, Vincent, Nicolas, Lionel, Victor are the names that first comes to mind.

I also think of all my friends in France and Turkey who were with me during this journey. I wish to thank them for the earnest moments we shared, full of fun and interesting discussions. Some of them even spared their time for proofreading this manuscript, a special thanks to them. Besides, I am deeply grateful to Cucus and PA for being there with me when it was most needed, and to Sinem for keeping on encouraging me.

At last, my warmest gratitude is to my family; my parents and my sister, who always supported me unconditionally.

Teşekkürler.

Chapter 1

Introduction

"I arise in the morning torn between a desire to improve the world and a desire to enjoy the world. This makes it hard to plan the day."

— E.B. White

Contents

1.1 Motivations

The general theme in which this thesis is positioned involves two driving forces of software development. One is about assuring the dependability of the software programs by making sure that they behave as predictable as possible [Laprie 1992, Zave 1997]. The other one is about making software evolutive and flexible enough so that it can react to change as fast as possible, even proactively. These two forces are usually incongruous with one another. The software engineering domain looks for solutions to reconcile them in the most efficient way, producing dependable software that can adapt dynamically to new conditions [Baresi 2010]. This section presents the motivations of this research, substantiating this point and thus forming the context of this work.

The success story of modern mobile devices such as smartphones and tablets is remarkable. In less than 10 years they now dominate the way users interact with computing services. According to the report [eMarketer 2014], the worldwide smartphone penetration has grown to 1.76 billion people in 2014, holding the 25 percent of global population. The reason behind this wild adoption is not the high-resolution touchscreens, integrated cameras or gyroscopes they embed. Notwithstanding the impact of those and many other technologies such as high-speed wireless broadband, mobile devices largely owe their success to the *"apps"* they offer.

The notion of application is most certainly not new. Back from the early days of computers, operating systems provided software stacks for applications to be executed upon. What smartphones (or tablets) did differently however, is to combine a pivotal physical interaction pattern – touchscreens – with development tools, SDKs and execution platforms destined for application developers. These software tools allowed third-party developers to easily program, deliver and execute their applications.

The, so called, explosion of *"apps"* happened thanks to this ecosystem where not only industrial software producers but also individual developers could develop and deliver their applications to the masses. For grasping the difference these tools make, it is sufficient to compare Java ME enabled SymbianOS phones with Dalvik VM based AndroidOS phones. Both platforms supported executing applications developed in Java, but clearly SymbianOS's application support was primitive, which lead to its decline as a platform. Indeed, developing applications for those platforms become so mainstream that the main challenge for developing a successful application become finding the right idea. Then the recipe followed by; backing it up with the right services and presenting it with a beautiful user interface design [Hitcents 2014].

The success of smartphone applications is just another example for use cases of domain-specific execution platforms. Another instance for such platforms is the frameworks for developing web applications. But a more important aspect for attracting application developers is providing tools to debug, test and deliver applications developed on these platforms. Mobile ecosystem providers such as Apple, Google and Microsoft all

achieved this by providing an *"appstore"*, an application market, helping users to discover and install applications on their device with one-tap. In spite of the fact that open source community repudiated these centrally controlled software stores at first, many software producers enjoyed being able to see their applications bought, delivered, deployed and run on consumer devices.

As a result, it is safe to say that what really propelled the mobile computing is the money vector provided from these mass-market retailing channels, i.e. application markets. The deployment feature (delivery, installation, activation) played a crucial role in implementing the infrastructures for *appstores*, therefore in the adoption of mobile computing.

In parallel to the advancements in mobile platforms, the Cloud Computing emerged as another aspect that marks the way applications are developed. For software producers, Cloud computing made provisioning virtual hardware resources and executing applications on those as easy as a calling a web service. Such that applications can be bundled in virtual machine images or lightweight containers and executed on the Cloud provider of choice. This type of Cloud called *Infrastructure-as-a-Service* reduces the cost of purchasing machinery, while providing great flexibility to lend additional resources to cope with increased demand. The *Platform-as-a-Service* providers such as GoogleApp Engine or Heroku rent ready to serve execution platforms to run applications while abstracting the underlying layers (i.e. the operating system, HTTP front-end). The complex task of administering the hardware and software infrastructures is therefore delegated. Similarly, these platforms enable deploying an application on multiple instances of an environment to better support the load. Although the Cloud reduces administration costs, the scalability is not guaranteed. Applications generally require to be designed with flexibility to make use of the Cloud. Yet, applications and businesses that can manage this evolution can harvest great benefits.

Cloud Computing, domain-specific execution platforms and tools for debugging, testing and delivering software all contributed to one final fact: Software is now developed in faster cycles. With better tools at disposal, the new norm in software development is to deliver a *minimum viable product* to customers as fast as possible. This allows development teams to learn about customer requirements and iterate to improve the product. Most importantly, automatized processes make sure that the software is tested, dependable and ready to be delivered to the customers as soon as it is produced.

Nowadays the computing world is sailing towards a next step with the emergence of a new class of tiny connected devices. In [Evans 2007], Cisco estimates that by 2020, there will be 50 billion *"things"* connected to the Internet, as opposed to approximately 10 billion this year. The growing interest for such devices is mainly due to the ability to gather information from the physical environment and control things through a software interface, giving the ability to automate this control. This automation paves the way for developing a new class of applications that leverage devices deployed in the real world.

Pervasive Computing envisions a whole new kind of relationship between computers and users. In this interaction computers are blended into everyday objects and users access information and use services without perceiving their existence. Put differently, the interaction pattern is not confined into a touchscreen of a mobile phone, nor a tablet or a television; but it potentially encompasses everything in the physical world that users interact with.

Currently, applications proposed in this domain do not exceed vertical, proprietary solutions. Clearly, as it was in the case of mobile computing, there is a lack of software engineering tools that help developers to program, test, debug, deploy and execute their applications. Dealing with real life environments, connected devices and human behavior, developing pervasive applications is difficult. The execution context of applications and the platforms that execute those, tend to evolve rapidly. This may be due to many reasons such as contingent devices, unstable network communication and the need for context-awareness. Among many challenges application developers and system operators face, the need to handle dynamism stands out to be the most eminent.

This work aims to improve this situation by investigating solutions for one of the software engineering problems, deployment of software, particularly in dynamic computing environments.

1.2 Research Challenges

This thesis investigates software deployment solutions for modern applications in dynamic execution environments. Additionally, it considers the requirements of pervasive computing for applying these solutions. Two actors are particularly involved and impacted by this work, namely, the application developer and the operator of execution platforms.

Four major challenges are addressed in providing the proposition of this thesis:

Heterogeneity: No two computer systems are bitwise identical in terms of configurations and capabilities. A software deployment solution must be able to target different platforms. Conversely, it must be able to ignore insignificant differences between platforms and customize the deployment for effectuating essential changes. Also different applications and platforms may need various kinds of configurations and actions to be performed.

Scalability: Applications and execution platforms can grow rapidly in size and complexity, incorporating high number of configurations. On the other hand, Pervasive and Cloud systems can require to run deployments on a high number of target platforms. The proposed solution must therefore scale horizontally and vertically.

Industrialization: To expect the same leap done in mobile computing from pervasive systems, the way software is developed and deployed must be industrialized. For software production, this means having automated processes for developing, building and delivering applications with a predictable and testable way. In addition, such systems must be cost efficient, promote productivity and robust.

Context-awareness: The deployment solution must turn the dynamically evolving nature of pervasive environments to its advantage by proposing policies that can be customized according to the state of the target platform and changes in this state.

1.3 Contribution

This thesis proposes a novel approach for deploying software in dynamic execution environments. This approach is based on a transactional, idempotent process capable of coordinating deployment actions. The properties granted to this process allow performing continuous deployments, in accordance with the current state of the target platform. Along with this process definition, this thesis proposes:

- the generic resource-based model in terms of which the platform state, the deployment request and the deployment process are defined.
- the deployment manager capable of continuously adapting applications according to the changes in the context.
- the domain-specific language for describing application deployments with architectural variability.
- the extension mechanisms for extending the capabilities of the deployment manager and the description language for handling new kinds of resources.

There are, however, some aspects that are deliberately left out of the scope of this work. Namely, the deployment in distributed computer environments is not treated in this work. The security aspect for transferring deployment artifacts is another subject that this work does not cover.

These contributions are developed and available as Rondo project. Rondo provides a tool suite containing, among others the deployment manager and the deployment descriptor language. These tools are fully operational and are tested against deployment scenarios defined within industrial and research projects, including pervasive platforms.

1.4 Dissertation Structure

After this introduction, the remainder of this document is divided into two parts, namely, the state of the art and the contributions of this work.

The state of the art includes three chapters:

Chapter 2 presents background information about pervasive computing. It examines the challenges brought by this computing domain, some of which this work contributes to tackle.

Chapter 3 studies the general concepts of software deployment. It discusses common issues addressed in software deployment and compares different deployment automation approaches.

Chapter 4 introduces the ideas and concepts behind continuous deployment. It presents current practices for implementing deployment pipelines. More particularly, this chapter proposes a characterization framework for evaluating continuous deployment solutions.

The contributions of this work are presented under three chapters:

Chapter 5 recalls the addressed problem, outlines the objectives and gives an overview of the approach of this work for enabling continuous deployment in dynamic execution environments. Then, it details this approach by presenting the formal framework, the deployment manager architecture and the descriptor language. It includes a series of discussions that these propositions invoke. At the end of this chapter, an evaluation of these propositions is presented.

Chapter 6 describes how the propositions of this thesis are implemented. Rondo is the tool suite which proposes implementations of the deployment manager and the deployment description language for OSGi™ platforms.

Chapter 7 presents experiments performed using Rondo tools for validating the contributions of this thesis. This chapter also reports the experiences of using Rondo for various deployment scenarios.

Finally **Chapter 8** concludes this document by summarizing the principal ideas and proposes future research directions revealed by this work.

Chapter 2

Pervasive Computing

"If history were taught in the form of stories, it would never be forgotten."

— Rudyard Kipling

Contents

2.1 Introduction

This chapter presents background information about pervasive computing. It describes the general idea behind this emerging computing domain, followed by motivating examples and various research domains that contribute to its realization. It continues by identifying general characteristics of pervasive environments. Following the context of this thesis, this chapter discusses requirements for developing pervasive applications focusing principally on middleware solutions. Relevant related work is presented focusing on middleware approaches. The chapter discusses limitations of existing work followed by a conclusion.

2.1.1 Evolution of Computing Environments

The world now is becoming increasingly digital, populated by a profusion of digital devices designed to assist and automate more and more human tasks and activities, to enrich human social interaction. However, this was not the case half a century ago, when analog machines of the industrial age left their legacy to the digital revolution, paving the way for miniaturized digital computers. Since the introduction of digital computers, computing environments have evolved constantly; thanks to technologies that allow increasingly smaller, more powerful, communicating and energy-autonomous devices to be built. Therefore, human perspective of computer systems has undergone different stages of evolution, each one altering the way humans interact with computers. The following analysis of this evolution is inspired by a similar analysis presented by Weiser in [Weiser 1996] which started with these introduction lines:

> "The important waves of technological change are those that fundamentally alter the place of technology in our lives. What matters is not technology itself, but its relationship to us."

The figure 2.1 illustrates the fundamental technological changes related to computer systems. It highlights some crucial steps that have marked its evolution. In particular, the notions of distribution, mobility and proliferation of computer systems clearly appear as major aspects. We examine more precisely in the following sections these different stages.

In the early 1940s, centralized computing was predominant and appeared as the only way to build computer systems. These took the form of isolated computers, requiring large amounts of space, even taking up whole rooms. They were compounds of processors and memory, and were administered continuously by experts. These experts were at the same time administrators, developers and users of equipment and software. These ***mainframes*** had limited resources and had to be shared among multiple users.

Later, with the evolution of electronics, the term *mainframe* was attributed to high-end powerful computers, running applications that serve a large number of users. Even

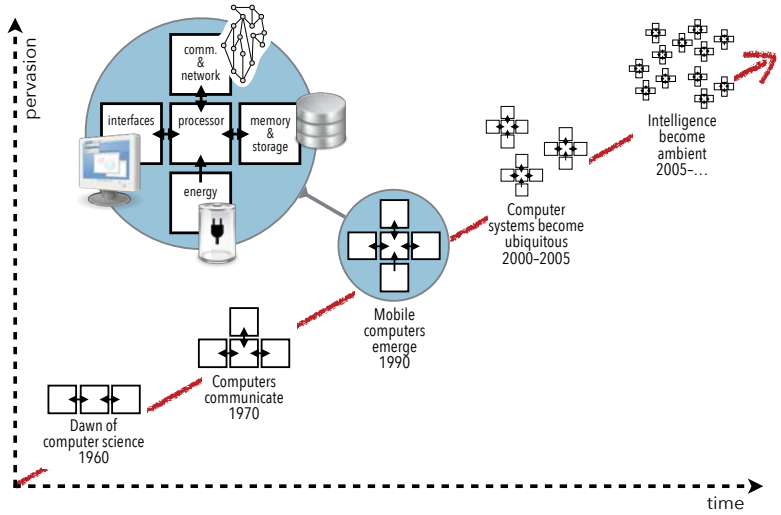


Figure 2.1: Evolution of Computer Systems (adapted from [Waldner 2007])

today, it is possible to have same kind of relationship of the mainframe era; anytime a computer is a scarce resource and must be negotiated and shared with others. Usually usage of domain-specific hardware with special calculation properties belong to that era of computing. In [Weiser 1996] Weiser summarized mainframe computing as

> "If lots of people share a computer, it is mainframe computing."

In this same analysis, Mark Weiser then introduced the emergence of personal computing as follows:

> "In 1984 the number of people using personal computers surpassed the number of people using shared computers. The personal computing relationship is personal, even intimate. You have your computer, it contains your stuff, and you interact directly and deeply with it."

Personal Computers (known as PC's) continue to be the significant way of human interaction with the digital world. This kind of interaction needs special attention from the user, as a user's principal intention is to use a service or access to information. As it is a personal belonging, most users are the administrator of their PC, installing and configuring the software developed and distributed by third parties.

While standalone PCs or mainframes restrictively use local resources, network infrastructures allow computers to access remote resources, interconnecting personal, business and government information. Computers with Internet access thus allow developers and service providers to build complex applications with more added value for users. However, they also created many issues for system administrators and application developers

such as distributed security, remote communication and integration of heterogeneous applications. To quote Wieser again,

> "Interestingly, the Internet brings together elements of the mainframe era and the PC era. It is client-server computing on a massive scale, with web clients the PCs and web servers the mainframes."

In early 1990s, emergence of portable laptop computers and wireless networks gave birth to mobile computing. It allowed users to access network-enabled applications while being mobile, therefore changing location. Moreover, increased attention to smartphones and tablet computers created a new kind of PC, that is powerful, mobile and connected wirelessly with high data-rates via new generation cellular networks (3G, LTE). These devices unfold new possibilities in terms of user interaction with computers; e.g. smartphones can be associated with their owner and they can be used to determine the position of the user via integrated GPS. Integration of mobile clients into existing distributed systems brought about new issues such as location sensitivity, energy-awareness and adaptive resource management [Satyanarayanan 2001].

In this post-PC era, more and more connected mobile devices dominate the human interaction with the computing world. Pervasive Computing, often also referred to as Ubiquitous Computing, is a vision for next-generation computer systems that are infused into real world environments. Pervasive Computing envisions a whole new kind of relationship between computing and users, exceeding mobile computing, where computers are blended into everyday objects and users access information and use services without perceiving their existence.

2.1.2 Pervasive Computing

Pervasive Computing, or as it was introduced in [Weiser 1991] Ubiquitous Computing, describes next-generation computing environments, which puts humans at the center of focus, rather than machines. The seminal paper of Weiser illustrates mostly perspectives of his vision for this new kind of human-computer relationship. Weiser and his colleagues in Xerox Palo Alto Research Center postulated a world saturated with tiny computing devices integrated into everyday objects and a computing infrastructure that interconnects these devices in order to support human tasks, in a way that all this is invisible to the users. Therefore users could concentrate on their tasks naturally, instead of worrying about how to operate the whole computing system.

Their ideas have inspired many researchers, which has lead to the appearance of different terms, such as ***calm computing***, ***disappearing computer***, ***everyware*** [Greenfield 2006], ***Internet of Things*** [Mattern 2005], ***Ambient Intelligence*** [Epstein 1998, Hansmann 2003] and ***things that think*** [Hawley 1997]. Although there has been a battle of concepts in media and the research community over the usage

of these words [Ronzani 2009]; basically all these terms point at an infusion of computing environments into the real world, following the vision of Weiser. The author of this thesis does not treat these terms differently, and in the context of this work, the term "pervasive computing" is used to refer to the general paradigm.

In the presence of these related concepts, instead of proposing a single definition of pervasive computing, one of the main goals of this chapter is to establish a common understanding of the vision it refers to. To this extent, it is crucially important to analyze different definitions in the literature:

> "The most profound technologies are those that disappear, [...] They weave themselves into the fabric of everyday life until they are indistinguishable from it." [Weiser 1991]

> "We characterized a pervasive computing environment as one saturated with computing and communication capability, yet so gracefully integrated with users that it becomes a 'technology that disappears.' [sic] " [Satyanarayanan 2001]

In previous definitions, authors take a bird's-eye view on the technology and focus on the seamless integration aspect of the services provided by pervasive computing. The following definitions focus more on the connectivity of different kinds of devices.

> "One could describe 'ubiquitous computing' as the prospect of connecting the remaining things in the world to the Internet, in order to provide information "on anything, anytime, anywhere. [...] the term 'ubiquitous computing' signifies the omnipresence of tiny, wirelessly interconnected computers that are embedded almost invisibly into just about any kind of everyday object." [Mattern 2001]

> "Pervasive computing calls for the deployment of a wide variety of smart devices throughout our working and living spaces. These devices are intended to react to their environment and coordinate with each other and network services. Furthermore, many devices will be mobile and are expected to dynamically discover other devices at a given location and continue to function even if they are disconnected." [Grimm 2003]

> "The basic idea of this concept [Internet of Things] is the pervasive presence around us of a variety of things or objects – such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, mobile phones, etc. – which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbors to reach common goals." [Atzori 2010]

From the definitions above, pervasive computing can be summarized into following core properties:

- Pervasive Computing is ***invisible*** through unobtrusive human-computer interaction.
- Pervasive Computing is inherently ***distributed*** among mobile and stationary devices, and network services. These devices are usually hidden from the user but constantly interacting with each other and their environment.
- Pervasive Computing is ***context-aware*** in order to optimize its operation to the current environment.

These core properties define briefly the pervasive computing vision. They reveal some of the fundamental aspects of pervasive computing like invisibility, distribution, mobility and context-awareness. However, they are widely incomplete to be able to uncover challenges for realizing the pervasive computing vision. Sections 2.3 and 2.2 of this chapter decomposes these core properties into detailed characteristics expected from pervasive computing system.

2.1.3 Context & Context-awareness

Context-awareness is an essential property for pervasive computing systems. Context-aware systems are systems that are aware of their "context" and that are able to adapt their operations according to the changes in their environment [Baldauf 2007]. "Aware" systems began appearing in the mobile computing era in the form of location-awareness. Location-aware mobile devices are able to determine a user's location and notify when the user changes their location [Bauer 2002]. Although location continues to be the principal context information, it does not necessarily represent interesting information for every kind of application scenario. Since then the way context information is defined has evolved towards more elaborate models.

Context-awareness was first introduced in the early years of pervasive computing, by Schilit and Theimer. The authors defined the context as the following definitions.

> "[...] the location of use, the collection of nearby people, hosts, and accessible devices, as well as to changes to such things over time" [Schilit 1994].

Later Dey proposed a general definition, which is accepted today as one of the most accurate definition.

Definition 1: Context

"Any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves." [Dey 2001]

Based on a general context definition, a classification is proposed for different types of context [Schilit 1994]:

- **Computing (Virtual) Environment** includes all the variables that describe the available computer technology such as used resources, available devices and resources, network bandwidth, etc.
- **User (Human) Environment** includes information on users: location, immediate needs, social activities, nearby people etc.
- **Physical Environment** describes the physical environmental attributes of the place the user is situated, including in particular, temperature, luminosity, noise level etc.

This classification of different types of context reveals an important concept about context-awareness in pervasive computing: It shows how pervasive computing brings together virtual and physical environments through a focus on users. In pre-pervasive eras, computing involved only virtual entities conceptualized by software developers. And the context was principally resources available to the computer hardware, which are usually virtualized by the operating system. However, pervasive computing involves not only an abundant number of computing devices but also users and physical environments. Context information about the users' situation and the physical environment are as important as the virtual context. Therefore, a ***pervasive computing environment***(or only ***pervasive environment***) consists of the intersection of these three environments (see figure 2.2).

Virtual, human and user environments are in constant interaction between each other. For example, the fact that a user changes his/her location causes the physical environment to change, which leads to changes in the computing environment via sensors. Or a change in the physical environment, like an increase in indoor temperature, can obviously affect user comfort but can also cause damage to the computer hardware.

Context-awareness is a central enabling technology for pervasive computing systems (see definition 2). It is required for creating computers and applications that are non-intrusive in terms of the user's perception. An important thing to note is that most of the research conducted in context-awareness is applied in pervasive computing environments. However the pervasive computing should not be reduced to context-awareness as it has many other requirements.

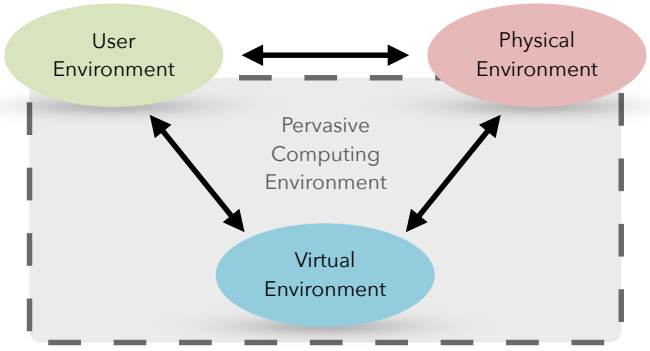


Figure 2.2: Pervasive Computing Environment

Definition 2: Pervasive Computing System

Aware of the changes in different contexts, a pervasive computing system coordinates the interactions between these environments, in order to provide useful services and information to users.

2.1.4 Motivating Examples

Pervasive computing postulates a world where people are surrounded by computing devices and applications that support and augment everyday activities. The focus is on developing pervasive computer systems to support people during their daily activities and tasks, to simplify these in a less obtrusive way. People will live, work, and get entertained in a seamless computer-enabled pervasive environment that is interwoven into the physical environment. A physical world integrated with computing devices and services have many implications for everyday life. This integration would change how people live their private life, how industries make money and how public institutions deliver their services.

a. Smart Spaces

Pervasive computing vision was deemed a futuristic but realizable one, especially while Moore's Law proved to be more accurate than its initial predictions [Moore 1965]. Early work in pervasive computing concentrated especially on integrating miniaturized computers into daily life and exploring new ways of human-computer interaction. These projects were the pioneers of pervasive computing applications. They aimed to implementing vertical application scenarios involving environments equipped with sensors, actuators and mobile devices called smart spaces [Kidd 1999].

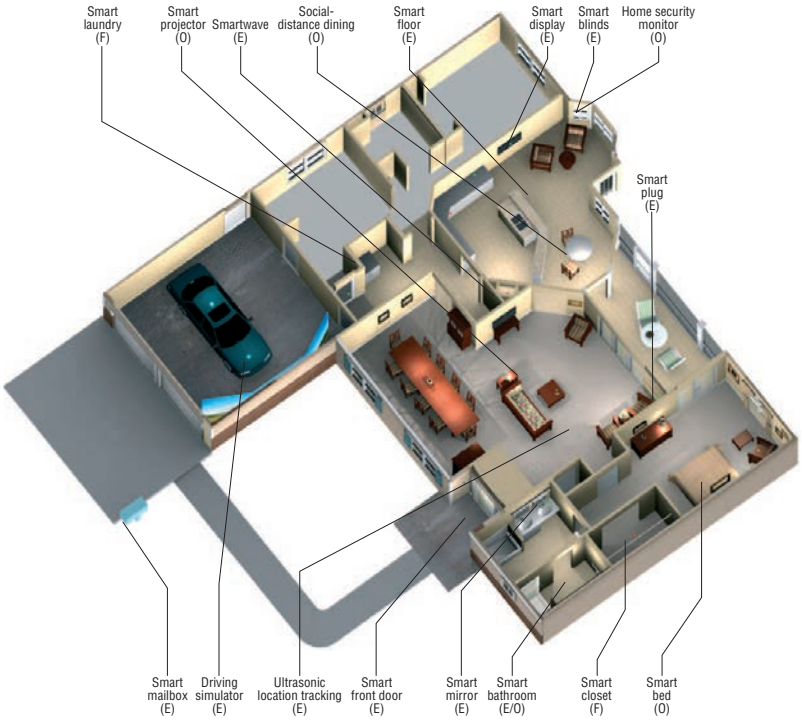


Figure 2.3: Example for Smart-space Environment (from [Helal 2005])

Location-aware office floors that forward incoming phone calls, context-aware meeting rooms that sense human activities, classrooms that are equipped with smart whiteboards and interactive surfaces, digitally augmented outdoors and yet wearable computers were the example of such application scenarios. Although they were mainly experimental prototypes, these early applications demonstrated some of the fundamental application areas of pervasive computing.

b. M2M Systems

Machine-to-Machine (M2M) systems are based on the communication between machines, without human intervention. In a M2M application, machines communicate with each other, using services of each other and exchanging data. These kinds of systems are widely used in different industrial areas such as environmental surveillance, logistics, utility infrastructures etc.

The idea behind M2M systems originates from the word 'telemetry', which means "measurements from distance" or "remote measurement". The concept of telemetry involves using sensors and remote machines for collecting data and sending it to a central location for later analysis. Types of machines interacting in such a system can range from tiny sensing devices that operate on low power, to powerful servers. Modern M2M systems bring considerable improvements over existing telemetry concepts. Wireless sensor technologies offer enhanced connectivity and sensitivity. Modern information systems

supported by interconnected servers and server farms enable fast processing of huge amounts of data.

M2M systems involve large-scale deployment of machines. Sensor networks connect mobile or (carefully placed) stationary sensor nodes that measure different metrics of their environment and send these measurements to centralized information systems fitted with databases and data analysis software. In this way, raw data produced by sensors eventually goes through a set of transformations and filtering, also known as ***data mediation***. Data, mediated to servers, then would be stocked in databases for later querying. They can be analyzed to create reports and to take decisions - either by humans or in best-case scenarios, avoiding human action by computer-based decisions.

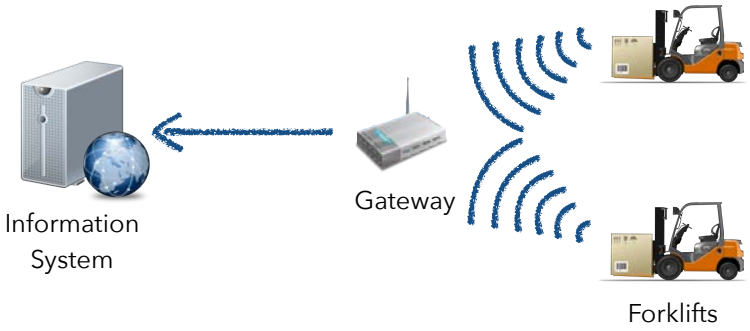


Figure 2.4: M2M Application Example (adapted from [Lalanda 2014])

M2M systems have many application areas including inventory management for retailers and manufacturers, water quality monitoring for public utilities, or even micro-weather forecasting for agricultural areas. The figure 2.4 depicts a M2M application for managing forklifts using wireless data transmission. M2M applications are about collecting and processing high amounts of data. In the context of pervasive computing, they demonstrate a special case of interaction between physical and virtual environments. Establishing software and hardware infrastructures to cope with this intensity of data, especially on wireless sensor networks, is extremely complex.

c. Smartlife

1990s witnessed widespread adoption of computerized information systems by most of the organizations of the modern world. Banks, retailers, manufacturers, press, utility/service providers, government agencies, almost every department in any company started using computer systems for automating and optimizing their business. Information, whether produced internally as a result of business processes or gathered from external sources; stocked in databases and analyzed, in order to take vital decisions about business management. As a result, information, and the information systems, became the most valuable business assets.

A decade later, with the prevalent usage of Internet in modern communities, these

organizations stawrted using the web for reaching their clients (or their users for non-profit organizations). Web applications allowed people to access information previously locked up in computer systems of these organizations. Mutually, for organizations, this meant a new playground for gathering information and improving their services and profit. Rapidly new kinds of businesses emerge from Internet-only services that leverage user-habits, usage statistics etc. Expansion in the usage of Internet services caused a boom in the volume of information stocked and analyzed in information systems.

In parallel, M2M systems let organizations expand the information gathering into physical environments, with the goal of monitoring and optimizing business processes. With the emergence of IPv6, sensors and actuators embedded in physical objects are connected through the same protocol that connects the Internet. M2M systems and Internet-based services became tools for harvesting information from the environments and users and communicating with them.

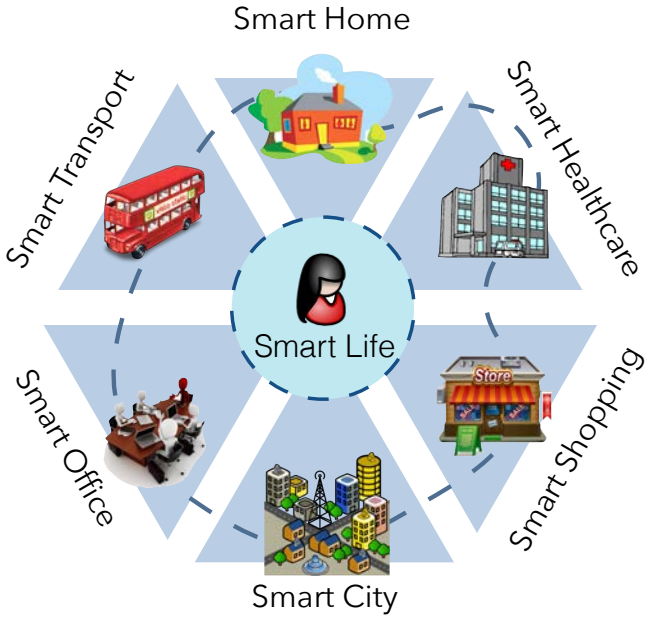


Figure 2.5: Smartlife Concept (adapted from [FP7 BUTLER Project 2013])

A term that is recently appearing in pervasive computing domain is 'Smartlife'. It defines a concept where organizations reach their customers and users in their everyday life by providing them useful services and information anytime, anywhere. It comprises different vertical pervasive computing domains like Home, Office, Transportation, Shopping, Healthcare, City and Utilities. On top of these existing domains, Smartlife proposes a new business model, which allows service providers and vendors to bring and integrate their services into these everyday environments with a holistic approach.

These new kind of services have the novelty of profiting from pervasive context-awareness. Applications and services integrated into 'Smartlife' use the extended knowl-

edge about users and their physical environments, and the ability to perform actions directly on user's environment via embedded devices. Usage of context information horizontally between domains allows an application, for example, to leverage information about the shopping list of a consumer in the context of home automation, to track the freshness of foods in the refrigerator and in the context of healthcare, to inform the consumer about their bad nutrition habits

The next step for the advancements in pervasive computing research will be the realization of Smartlife concept. This will require the contributions from smart-spaces, mobile computing and M2M systems. Work on smart spaces would provide more natural, ubiquitous interaction with users and their environments, whereas M2M systems offer the information systems for enhancing this interaction with business services. This points out to a need for developing hardware and software infrastructures that will host and execute these kinds of pervasive applications with specific requirements.

2.1.5 Research Domains

Pervasive computing gives rise to new challenges in different domains. Attaining this vision requires collective research efforts in a variety of areas, including microelectronics, telecommunication, embedded systems, wireless networks, information systems, software engineering and also social sciences.

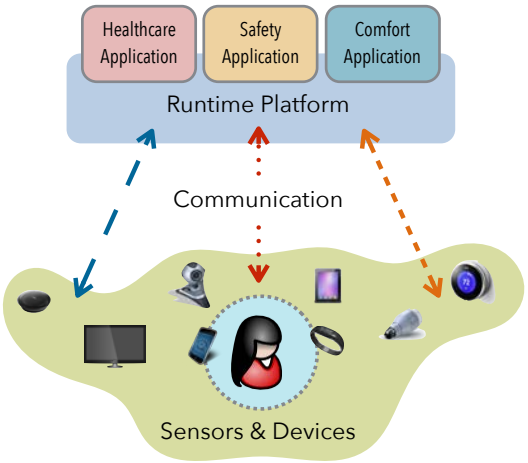


Figure 2.6: Pervasive Computing Technology Stack

Figure 2.6 illustrates different levels of domains contributing to the vision of pervasive computing, as sensors and devices, communication protocols, runtime platforms and application development. In this section some of the research advancements that have a direct impact on pervasive computing are introduced.

a. Smart Devices

Advancements in microelectronics and circuit design allow to produce more powerful and smaller embedded devices that can integrate better in physical environments. With the usage of advanced materials and production techniques, it is possible to produce devices at low costs. Low device costs and wireless communication technologies enable large scale, nomadic usage of self-powered devices. However, improvements in these aspects are not backed-up by efficient energy capacity solutions. Internal power sources, such as batteries, constitute the primary limitation for size, weight and lifetime of current devices.

Many efforts on creating smart devices are concentrated on increasing energy autonomy. With reconfigurable hardware and software solutions, embedded devices can reduce their energy consumption by optimizing their operation (usually less wireless communication means less energy usage) [Druilhe 2013]. In addition to that, energy-capturing solutions such as photovoltaic cells, piezoelectric modules or radio wave energy harvesting antennas allow devices to convert environmental energy into electrical power [Sudevalayam 2011].

In addition to energy optimization, there are other challenges for manufacturing smart devices that integrate and interact better with physical environments and users. Usage of innovative technologies like nano and organic materials opens the way for constructing little devices with precision that can be embedded even in the human body [Ratner 2003]. In the light of these advances, in the near future it is predictable to see devices that are negligibly small and that have greater energy autonomy.

b. Wireless Communication & Networking

Most of the recent smart devices benefit from the advantages that cable-free wireless communications offer. Wireless communication technologies ensure smart devices' connectivity to the outside world, while freeing them from constraints of wires, enabling mobility and widespread deployment. Wireless networking solutions are needed especially in sensor networks where sensor nodes are spatially distributed and interconnected via wireless communication. Nodes transfer data and measurements from one node to another until it reaches a base station, or a gateway. In the gateway, these measurements can be exploited by applications and presented to users. This way, sensor nodes can be deployed within longer ranges and still be connected to the gateway.

Wireless technologies bring new problems in terms of communication quality. The research community's efforts concentrate on developing hardware and networking solutions to provide the best possible communication quality while optimizing a device's functionality and battery life. Innovative networking solutions offer adaptive network topologies that self-optimize and self-repair in case of bad connections between nodes or non-responding nodes [Dijkstra 1974]. On the other hand, application level communication protocols are emerging that are tailored for lossy, low-bandwidth networks, such as CoAP [Shelby 2014] or MQTT [Hunkeler 2008].

c. Software Engineering Processes & Tools

Regarding the advancements in microelectronics and wireless communication, it becomes possible to leverage the capabilities of smart, wirelessly communicating devices for developing pervasive applications. However, there are still some key challenges yet to be addressed. A major challenge for pervasive service providers is dealing with the high complexity of development, integration, deployment and management of pervasive systems and applications [Schiele 2010]. Research in software engineering strives to come up with solutions that will ease the development and execution of pervasive applications.

Middlewares aim to provide a solution for easing the development and execution of applications. They stand between operating systems and applications to provide high-level abstractions and functionalities, hiding certain complexities of application development. They have evolved from simple technologies hiding network details of distributed applications into important blocks of software that hides and deals with many aspects such as heterogeneity, mobility, data processing and scalability. Throughout their evolution, middleware systems adopted and enforced software engineering principles such as separation of concerns and modularity to manage the increasing complexity of applications and facilitate programmability. Moreover, recent works on middlewares concentrate on providing runtime support for monitoring and managing applications during their execution [Floch 2006].

In addition to middlewares, development tools also provide ways to facilitate software development. Build automation tools help streamline compiling and packaging stages, and ease the distribution of software artifacts. Dependency management tools such as Apache Maven serve to manage complex projects with multiple dependencies, promoting the modularity in software development. Integrated Development Environments (IDEs) are software programs that usually include different tools such as editors and building tools for generating executables and debuggers for testing. The popular IDE, Eclipse, provides an extensible environment, which can integrate different tools for modeling, programming, dependency management, building, versioning, testing, etc. With its plug-in system, Eclipse allows for integrating new tools and provides an ecosystem for building domain-specific IDEs. For instance, Xtext allows creating domain-specific languages and corresponding code editors based on Eclipse.

Integrated tools ease the efforts of application designers and developers to create testable and maintainable applications. In addition to that, most of the existing tools constitute a solid base for extending these capabilities to specific domains, in which it is particularly difficult to create applications. Pervasive computing is one of these field, where domain-specific tools address specific issues. The section 2.4 of this chapter presents different types of tools in more detail.

d. Social Sciences

The inevitable emergence of pervasive computing raises issues beyond technology and operating techniques. It requires sociological and philosophical studies on the understanding, acceptance and usage of possible pervasive applications [Bohn 2005].

Before being adopted in a widespread fashion, pervasive computing systems need to be accepted by the ethical barriers of the society. The prevalent infusion of connected computer systems into every aspect of life can be unnatural for some of the age generations. Current discussions over the privacy concerns of social networks provides a good example for what may be the struggle for pervasive computing.

Another societal concern involves the government's implication and regulations over such computing systems. While modern open societies accept the governments' role of regulating certain domains in order to protect individuals' civil and property rights, recent experiences shows that the same trusted organizations can be transformed into mass violators of human rights.

Although these research themes are well beyond the scope of this thesis, they are nevertheless inseparable from the consequences of the pervasive computing vision.

2.2 Characteristics of Pervasive Environments

Pervasive computing is about uniting physical and computing environments with the intention of providing human-centric services. Pervasive computing is more of a vision about the evolution of computing than a whole new domain. It is difficult to reconcile different definitions, to have an exact definition of pervasive computing (see previous section 2.1.2). However, it makes sense to define its properties that lay out its goal and the challenges between. This section details important characteristics of pervasive computing environments.

2.2.1 Distribution

Services and information offered to the users in pervasive environments often originate from different remote providers and sources. Some of these service providers and information sources consist of embedded devices that are dispersed (hidden or exposed) in physical environments, such as environmental sensors, mobile phones, electronic appliances or actuators. These devices are constantly in interaction with physical and user environments – they are capable of sensing their environment and acting on it; they incorporate user interfaces that let users interact and access information. Resources on these devices are accessed using different communication protocols, based on either wired or wireless technologies. Because of their limitations in terms of computing power and stocking capacity, applications using capabilities of these devices and coordinating them, do not

necessarily run on top of these devices. This particularity makes pervasive computing environments inevitably distributed.

Resources available for pervasive computing are generally not limited to devices present in a physical environment. For instance, in M2M applications, measurements collected from sensors are sent –via Internet– to remote servers for stocking and analyzing the data. These information systems, usually highly capable in terms of computing power and stocking space, are located physically too far apart to be called *pervasive*. Nevertheless, their resources can be leveraged in pervasive computing by providing value-added services and information, that otherwise wouldn't be possible.

The increasing number of communicating devices and servers creates the need for large-scale deployment, installation and maintenance of software and hardware components. Computing infrastructures in pervasive environments (both hardware and software) should have scalable architectures to cope with high density of devices, provided services and produced data. Software designers and developers should take into account the distributed nature of applications, remote services and devices.

2.2.2 Heterogeneity

Every year there are more and more device manufacturers offering products for usage in the pervasive computing domain. Communication protocols are equally diversified, as each device type has different characteristics and requirements regarding the nature of its use. Many working groups have made efforts to standardize common protocols. UPnP, Bluetooth, Device Profile for Web Services (DPWS), ZigBee, X10, KNX are just well-known examples. However, industrial device manufacturers usually prefer using proprietary protocols instead of sticking to standards. They want to keep their product environment private and closed in order to sustain their product ecosystem and continue selling products. Because of this it leads to a device market that is highly heterogeneous.

In pervasive environments accessing and using resources on heterogeneous devices like sensors or actuators is only one side of the problem. It is equally important to administer and configure devices present in an environment. Different manufacturers are likely to use different device management models and protocols to represent information about the device itself and perform maintenance functions. A similar tendency is seen in web technologies for service models. Accessing services over web, and exposing functionalities as remotely accessible services, requires integrating different service models along with communication protocols.

As a result, application designers have an increasing need to integrate new communication protocols, device types and services into their applications, which without a doubt increases the level of complexity of an application. Many pervasive frameworks are supporting only a limited set of protocols such as [Helal 2005] and [King 2006]. To meet the market evolution, pervasive platforms must support an open set of protocols, which can

be extended after the initial deployment of the platform and applications.

2.2.3 Openness & Plural Authority

In a pervasive environment, computing resources, either in form of devices or services, usually belong to different stakeholders such as device manufacturers, vendors or service providers. Applications running on top these environments need to interoperate these resources. This is only possible in an open world (open environment) where each one of these devices and services are designed to be open i.e. all or some of their functions are accessible *openly* by other devices or applications. Although openness is a prerequisite for creating pervasive environments, many systems today are still designed to restrict openness and interoperability. Vendors may deliberately restrict openness and ignore interoperability with a competitor's services, in order to preserve their market share.

The restriction of openness implies that the access to the resources of some devices or services may be subject to constraints and authorization. In pervasive environments, multiple pervasive applications run in the same environment. They access devices available in the environment at the same time, sharing their resources and functions. Applications can eventually have different levels of authorization of access to these resources. For instance, a certified application from a device vendor can have full access to its own devices, while another application would have limited access. These kinds of restrictions may serve device vendors or service providers to keep a certain level of control over their products, while continuing to contribute to the open environment with publicly available services. As services can have access to personal data, another reason for these restrictions is the privacy concerns of users. Maintenance and management of such an environment is complicated where multiple applications access shared resources (from devices or services) and interoperate with each other.

2.2.4 Dynamism

Evolution is an essential property of every computing environment. Hardware components fail due to faulty electronics or environmental conditions, and software programs have bugs that need continuous maintenance and updates. In open environments evolution is prevalent. Every device, every system that contributes to the open environment evolves and changes through time. Pervasive environments are an example for such open environments. Pervasive applications access and coordinate the resources from these systems; using services from remote systems or devices present in the physical environment. Openness allows pervasive applications to dynamically discover new resources and use them, while the state of previously available resources may degrade and become inaccessible.

In addition to being open, pervasive computing environments are constantly in relation with physical and user environments. Because of continuous changes in user and

physical context, elements in pervasive computing environments are forced to evolve dynamically. Changes in these environments, combined with open world assumption, there are many reasons for the pervasive environment to be dynamic, including:

- **Service availability:** In an open and heterogeneous environment, there is a high chance that frequently; a pre-known resource is not available, because it undergoes a software update or system maintenance, triggered by users or service provider administrators. Yet another reason that would undermine availability is limited device resources, so that it does not allow simultaneous access to its services.
- **User mobility:** Users move freely in physical environments, whether indoors or outdoors. The mobile devices carried by users also change location with them. For example, services on a Wi-Fi enabled smartphone are accessible when the user carrying it enters an area covered by a Wi-Fi router. Likewise, as the user exits the area, devices signal coverage would drop and the services on it will be in unavailable.
- **Device contingencies:** Devices that are designed to be used in pervasive context are generally low-cost and unsturdy. They are designed to maximize usage time with minimum resources. In some cases, like RFID tags, they are even disposable and negligible. Device functionalities and communication capabilities are usually affected by physical properties of the device and its environment such as heat, radio interferences, and battery level that lead to errors or unpredicted behavior.
- **Users' interaction with the environment:** Users' interaction with their environment allows applications to gather information about users intents and actions. Context-awareness of pervasive applications depends on this interaction. Sometimes users can cause devices not to work, by turning them off.

Because of the dynamism in pervasive environments, applications hardly find all needed resources that were on the whiteboard at the design time.

2.2.5 Autonomy

The pervasive environment consists of transparent relationship between users, physical environments and computer systems. To hide the complexity of the entanglement between the physical and logical world, it is necessary that pervasive computing environments be as independent as possible. The autonomy of pervasive environments is crucial particularly for two reasons.

First of all, in order to satisfy the seamless integration into real environments, interactions between users and pervasive systems must be natural and transparent. The user can know that he is dealing with an augmented physical environment, but he/she must not have to worry about how the system works. To guarantee an always usable system, in face of changes in the involved environments, the pervasive system must adapt in

response to these changes or even anticipate the change. The more a pervasive environment is autonomous, the less its users will need to adapt their behavior to interact with it. Autonomy is thus a characteristic, which promotes interaction transparency between humans and pervasive systems.

Secondly, the large-scale deployment and adoption of pervasive systems brings a problem of administration and maintenance. As these systems are situated in heterogeneous, open and dynamic environments, their management requirements are greater. Moreover, the physical environemnts in which pervasive systems are usually installed are not always accessible by system administrators. The autonomic management of pervasive systems is therefore essential, in order to alleviate the burden for system administrators maintaining these systems.

2.2.6 Summary

Pervasive environments are by nature distributed, heterogeneous, open, dynamic and unpredictable. Within these environments, a multitude of actors and entities are interacting in a natural and transparent manner. These are situated in and integrated with real physical environments, of which the boundaries can be precise (e.g. inside a building) or sometimes very blurred (a park, neighborhood, city, etc.).

All the characteristics presented above are more or less intended by the vision of pervasive computing and its various interpretations and applications (such as smart spaces and smartlife). They express how pervasive environments are perceived from the outside: how it interacts with the physical elements, with users and/or other systems.

These pervasive properties have a significant impact on the systems and applications that are contained and are operating in these environments. A computer system is composed of both hardware (sensors, displays, peripheral devices, etc.) and software (data, applications, components, ...) elements. According to the context and its use, applications coordinate these elements , in order to provide functionalities they are designed for.

In the case of a pervasive environments, in addition to the properties of which have been stated above, applications will need to handle some essential aspects. The next section presents how pervasive computing systems and applications are distinguished from traditional systems, by taking into account the characteristics required for pervasive computing.

2.3 Characteristics of Pervasive Applications

The application software, or just ***application***, is software that performs specific tasks for users. Applications are generally installed on top of system software that operates hardware and manages access to resources. In the case of applications running on mid-

dlewares, the middleware sits between the application and the system software, offering a more convenient, managed way to develop and execute applications.

Whether running directly on the operating system or on top of a middleware stack, an application's design and techniques used during its conception are strictly linked and sometimes constrained by the underlying systems capabilities. The above-mentioned characteristics of pervasive environments impose new challenges to the existing techniques employed for creating pervasive applications. For a better understanding of the features that will facilitate the conception of pervasive applications, one should look at the requirements of the applications running in pervasive environments. Following are some of key properties that separate pervasive applications from traditional ones.

2.3.1 Resource Management

Traditional applications are conceived to work with a set of predefined resources. Either running locally or distributed among distant machines, traditional applications are owned by devices (machines), and are restricted by the resources these devices provide. However, in recent years, there have been several movements that changed this paradigm. Especially the emergence of personal mobile devices opened a new era, where computing has become more and more human-centric. Pervasive applications conform well in human-centric vision of computing, where applications are associated to users and places rather than devices. Consequently, these applications need to discover, manage and use different devices and heterogeneous resources.

Traditional systems such as PCs or enterprise servers execute applications with a set of resources that are predefined and abstracted by operating systems (OS) or middlewares. CPU time, memory, disk space, network bandwidth are examples of such resources. These systems use abstractions to simplify the access of applications to resources. For example, OSs provide filesystem abstraction to manage disk access requests from applications (processes). Through this abstraction, it is also possible to manage the access authorizations. In [Krakowiak 2007], a resource is characterized by a number of properties, which impact the way it may be used and managed:

- **Exclusive or shared:** The resource may be exclusive to a particular application or simultaneously shared between multiple applications.
- **Stateful or stateless:** The resource may have a state related to the application that currently uses it.
- **Individual or pooled:** The resource may be individual or may be a part of a pool of identical resources.

In addition to these properties, a resource may have other attributes such as its location and may accept a number of configurations that will affect its behavior. All of the

properties and configurations are usually expressed in resource descriptors, which are communicated during resource discovery.

Devices in pervasive environments are typically first class resources for applications. Device functionalities can be shared between applications, or more critic functionalities can be exclusive to applications with specific permissions. An example to such operations is device configuration: While most of the devices function in stateless mode, they are becoming more and more configurable. Configurability enables optimization of device functions according to the changes in its condition (e.g. battery level). However, a change in device configuration inevitably affects all applications that use the device. Heterogeneous devices and protocols complicate virtualization of access to resources, thus making them individually managed resources.

It is a challenging task to handle access permissions, fair use of devices and coherent device configurations at the same time; in environments with high number of devices. Middlewares and OSs already incorporate some management policies for resource access [Bernstein 1996]. Beyond that, dynamic and unpredictable nature of resources in pervasive environments requires adapting and rethinking these policies.

2.3.2 Data Orientation

Pervasive applications offer services with added value by leveraging the data gathered from different sources, including sensor devices. So it is only natural to expect that in a pervasive application, a service depends not only on other service specifications but also on well-defined data types, where meta-information of the data is more important than its origin. Also, this data-orientation imposes a programming scheme where the consumer reacts to an event containing data produced by the provider. Therefore, a pervasive middleware should enable defining dependencies over data types and assure that these dependencies are satisfied with the data produced by data provider services.

2.3.3 Notion of Context

As discussed earlier, context-awareness is one of the core properties of pervasive computing. The context may be any information that is relevant for the application and it can be separated into three groups as user context, physical context and execution context. But the concept itself is not new, because the latter has had its place in applications for a long time, since developers need information about the state of underlying system – hardware or software. The need for context modeling became more apparent with programs that were executed by virtual machines such as Java. Even though WORA (Write Once, Run Anywhere) principle reduced development efforts of cross-platform programs, as different platforms can still exhibit different behaviors, the developers should take this into account in their code. A very simple example of a way to access the 'context' of an underlying virtual environment is through the system properties in Java. It lets developers

access primitive static information about the OS version, OS architecture, etc.

Determining user context in application code is a bit trickier. In traditional applications such as web applications, users change their context much more frequently than in the virtual environment in which applications execute. In this case, a user's context can be the browser used to access the web page, the visiting history, the cookies and so forth. The popular Servlet API [Sun Microsystems 2013b] was introduced in the early years of Java provides standard mechanisms to represent an HTTP request to a server. It allows server-side developers to access information about the request, and therefore construct a user profile that represents its context.

As for pervasive applications, in addition to virtual and user context, they are also involved with physical context. Pervasive applications need to transform raw data sensed from devices (measurements from sensors, indicators from other appliances) into more meaningful state indicators about the physical environment. Determining complex user context (e.g. behavior, mood, intention) and the dynamic virtual context (e.g. availability of resources, performance metrics) is more difficult in comparison with traditional applications. In many cases the content of the context is very subjective to a particular application. Therefore, applications incorporate "context provider services" that are responsible for transforming raw information from different, possibly heterogeneous sources to context state [Huebscher 2004]. For the sake of context-awareness, middlewares should employ mechanisms to inspect the virtual execution context. Moreover, providing support mechanisms for applications to construct their formal context models would not only decrease development times but also enable context-awareness at runtime.

2.3.4 Adaptability

Context-awareness requires that pervasive applications adapt constantly to the changing context. Pervasive applications should continue to satisfy user requirements in face of contingent devices, failing software modules and in general continuously changing context. In order to do this, it should be aware of its context and flexible enough to be able to apply necessary configurations and change its behavior. In addition, all this adaptation should take place autonomously to reassure user acceptance and fulfill the pervasive computing vision. Most of the traditional applications are developed to fulfill a fixed set of requirements. However, in pervasive environments, due to dynamic context, there are variations in requirements that may not be covered with a static application [Hallsteinsen 2012]. Therefore, fully specified, statically coded applications are not a good match for pervasive environments. Applications should be developed and executed with regard to these possible variations in requirements. On one hand, at design time, developers need to specify and develop the system providing different configurations of their applications. On the other hand, during execution, applications should be composed in a flexible manner allowing dynamic reconfiguration, meaning dynamically passing from one configuration to another. A typical example in mobile pervasive envi-

ronments is the case where availability of a certain device triggers the change: According to the location of the user, the application may choose to display its user interface between a high-definition screen or a portable device, optimizing the amount of information shown to the user.

Moreover, there may be different variability choices within an application. These choices should be coordinated in order to provide optimal operation in a given context. Due to rapid changes in pervasive environments and the lack of human administers, need for autonomic approaches emerge to guide dynamic adaptations at runtime.

2.3.5 Security

As discussed earlier, security mechanisms are needed to control the access of applications to the resources. Authentication, authorization, and accounting (AAA) protocols may be implemented on different layers of the pervasive system, including the middleware, in order to control resource access, enforce permission policies, audit resource usage, etc. Another important aspect concerns privacy in pervasive environments. Gathered data from various devices may contain or be used to deduce private information about the users' life. In the presence of multiple applications and devices from different owners, middlewares need to preserve users' privacy.

One of the main challenges for establishing security mechanisms in pervasive environments is determining user identity. Usually it is not possible to ask users to identify themselves as in web pages, and therefore applications should be authenticated with different credentials (e.g. platform owner, application owner) and handle secure communications permanently.

2.3.6 Summary

The development of pervasive applications raises particularly difficult challenges, much more demanding than those encountered for traditional applications. As a result, developing pervasive systems and applications requires a very high skill level from developers, far beyond what is usually encountered.

It is therefore necessary to provide specific tools that ease certain tasks during the design, development, deployment, execution and maintenance of pervasive systems. The purpose here is to abstract a number of problems such as those mentioned above: adaptability management, data management, security management, etc. Many works in this direction have already been completed, with varying levels of success. The next section explores a set of tools that contribute to the development of pervasive applications.

2.4 Building Pervasive Applications

The above points served to identify the founding principles of pervasive computing. These guidelines define a number of essential characteristics of pervasive environments. In order to preserve the nature of such environments, pervasive applications must show a specific set of properties, which allow seamless integration into the environment.

Consideration of all these properties has a significant impact on the lifecycle of pervasive applications. This section focuses on trying to understand what are the effects of pervasive properties on the lifecycle of applications, what new challenges it brings to their design and execution, and finally what are the software engineering approaches that allow easier implementation of these applications.

Since the dawn of computing, computing systems have become more sophisticated and software programming is becoming more and more complex. In the late 1960s the discipline of ***Software Engineering*** has emerged as a response to this increasing complexity. The period following the beginning of software engineering discipline, has witnessed the falling prices of computer hardware and the miniaturization of computers, which led the way to the era of personal computing. Until then computers owned by large institutions (governments, universities, private industries, etc.) were programmed and maintained by same people, and the software they execute was custom developed for these systems. The way of operating-in-isolation allowed strict control of the lifecycle of software development until their execution, and their distribution was very limited or non-existent.

The personal computer era has completely changed this mode of operating. Along with computers, software has been distributed to large user communities, who became the de facto administrators of their machines. In addition to that, a multitude of new concepts emerged, which increased the size and complexity of software programs: graphical interfaces, multi-user, concurrent programming, etc. Clearly empirical, ancestral methods previously used to design and execute programs were not suited to meet these new challenges. Software was delivered late, costed more than expected, was unreliable and/or inefficient.

It was in this time of crisis, which is now called ***software crisis*** that software engineering appeared, offering systematic methods for designing and implementing software. The employment of these formal or semi-formal approaches has helped build large projects, resulting in reliable programs and predictable delivery times, in accordance with the fixed costs of production. The techniques developed by this new branch of computer science have overcome the software crisis, and paved the way for the wave of personal computing and waves that followed until the pervasive computing.

To facilitate software production throughout the lifecycle of the application, software engineering offers many tools and methods: requirement analysis tools, compilers, code interpreters, shared libraries, dependency management tools, testing tools, deployment tools, code complexity analysis, monitorung tools, etc. The knowledge in the software

engineering field is vast and varies according to the approaches used to develop the software and to the concerned phases of the application lifecycle. Early during the emergence of pervasive computing idea, researchers worked on identifying requirements of building applications for this new field [Banavar 2000].

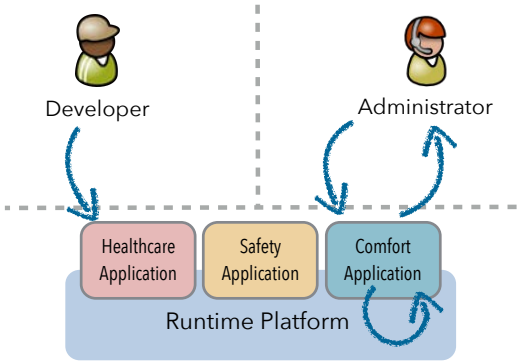


Figure 2.7: Application Tools

Various approaches can be divided into three families of tools that ease the conception and execution of applications: development tools, runtime tools and management tools (see figure 2.7). All three have the same goal, namely to shorten the lifecycle of applications, making their design, implementation and/or their maintenance easier, faster and cheaper. Even so the approaches taken by these tools are very different, each one focusing on a well-defined part of the lifecycle. ***Development tools*** focus on the design phase of the software by providing support mainly to the developers. ***Runtime tools*** such as middlewares are placed in execution between the target software system and the application, providing an abstraction for a simplified execution. And ***management tools*** such as monitoring and deployment tools focus on correct execution and the maintenance of software systems. In the following part of this section these three types of tools are presented more in detail.

2.4.1 Development Tools

In software engineering, the first approach to simplify the development of applications is based on providing a variety of tools for the development activities. These tools are therefore designed to support the maximum effort during phases of application lifecycle, reducing the task of the developers and also human errors. Some of these focus on how the software will be designed by simplifying the code to produce. They generally offer a development model that overcomes intricate details, such as the hardware architecture, memory management, and communication protocols. This type of development tools includes ***programming languages***, ***compilators*** and ***debuggers***.

Other tools alternatively focus on the project's infrastructure, facilitating the workflow of development teams. If they do not participate directly in the development of the

final product, but they greatly facilitate its development, construction and / or maintenance. These tools are, in particular the ***version control systems***, ***bug tracking and issue management systems***, ***production engines and project management tools*** and ***code analysis tools***.

Finally, some of the works have a more holistic approach and try to cover a large part of the lifecycle of the application. They bring together the tools described above and integrating them into a unified environment, even in some cases until the execution of developed applications. The approach of Computer Aided Software Engineering (CASE) tools designed to bring together project management software environments, ergonomic and have an overview of the project throughout various phases of lifecycle.

2.4.2 Runtime Tools & Middlewares

The word ***middleware*** is a generic term designating an intermediate software layer that sits between computing resources and manages one or more applications [Krakowiak 2007]. This intermediate layer is to facilitate access to these resources, and thus to simplify the execution of the application, which may have positive impacts on the design, development and/or deployment.

The concept of middleware has appeared during the emergence of distributed computing. The main problem at the time was that the systems could not communicate naturally together because of their differences in architecture and communication protocol. The solution has been to place an intermediate layer that abstracts the differences in architecture and protocols, and undertakes to translate exchanges between heterogeneous systems. As system complexity increases, areas covered by middlewares are expanding. In addition to the management of distributed communication, middlewares provide other functionalities such as ***distributed naming service***, ***data persistence***, ***transactional operations*** and ***runtime management*** and ***monitoring*** of applications.

The founding principle of middlewares is thus the abstraction layer: applications use the managed resources through a model defined by the middleware, which hides the complexities of management of underlying resources. Technical aspects for managing this complexity are provided by the middleware, and not required to be integrated into each application. Providing these common functionalities is often complex and prone to many errors. The code provided by the middleware, that implements these technical aspects, becomes non-functional from the point of view of applications, and is not related to the business domain of applications. Applications completely delegate these technical aspects, and ultimately eliminate much of the potential sources of error.

There are lots of research projects that address the challenges of pervasive computing through middleware solutions. Indeed regarding their level in the software stack, it is logical and versatile to employ middlewares to resolve most of the problems introduced in this chapter. Gaia [Román 2002], Oxygen [Rudolph 2001], Aura [Garlan 2002], MU-

SIC [Rouvoy 2009], DiaSuite [Cassou 2010], WComp [Tigli 2009], Base [Becker 2003] and PCOM [Becker 2004], PLASTIC [Bertolino 2009], Atlas [King 2006], HealthOS [Lim 2012] and PerLa [Schreiber 2012] are only some of the examples of such middlewares. The goal and the scope of this chapter and this thesis is not to provide a comparative study of capabilities of these middlewares but to point out their importance in pervasive systems.

In addition to those, there are generic execution platforms that are used commonly in the pervasive context. Fractal [Bruneton 2004], K-Component [Dowling 2001], Kevoree [Fouquet 2012] and Apache Felix iPOJO [Escoffier 2008] are some of the component-based execution platforms that provide the basis for constructing frameworks and middlewares.

2.4.3 Management Tools

Middlewares provide useful abstractions that facilitate the management and supervision of executing applications. But still tools are needed to help system administrators and operations teams to install and supervise computing systems. Especially in the pervasive context, the above-mentioned characteristics aggravate the inherent difficulty of applying management actions on these systems. Despite the unpredictable nature of pervasive computing, the administrators need to ensure the reliable execution of runtime platforms and of applications on top of those. The management domain can be studied in three categories as ***deployment***, ***monitoring*** and ***administration***.

The administration of a system involves mostly the configuration of hardware and low-level software stack of computing systems. The administration of large-scale, distributed systems is already an issue addressed since distributed systems. The heterogeneity and openness of pervasive systems adds new challenges to the mix. In such a system, the number of types of actions and configurations is high and has the possibility to increase. The autonomy requirement of pervasive systems impacts the way they can be administered. That is why automation is needed for configuring multiple machines. TR-069 [Broadband Forum 2013] is a commonly used protocol in telecommunications industry for administering devices connected to Internet.

The deployment process involves the sequence of actions that brings software from development to execution. Although most of the times the deployment is used to refer to the first installation of software to a administered machine, it is not restrained to that. It includes the process that changes the software at the target environment, with updates, reconfigurations and eventually the uninstallation. Deployed software can be a single application, multiple applications in the same time or the whole runtime platform. As the deployment process involves several complicated actions on the target machine, its automation is equally crucial for ensuring the correctness of the system. Requirements related to the dynamism are especially challenging for deployment of pervasive systems.

Among other challenges, the deployment process must ensure the adaptability of the software according to the changes in the context. The software deployment constitutes the main subject of this thesis and the following two chapters present the software deployment domain in more details.

The monitoring of computing systems is essential for tracing the evolution of the system and produce useful feedback on problems of software and hardware. This is enabled via *sensors* that are carefully placed on the software system, collecting the information produced and reporting those for analysis. The analysis of monitoring data can involve detecting correlations and calculating business metrics. In pervasive systems, monitoring must include the pervasive context, i.e. the information that the system has about its physical and user environment. In addition to that, the dynamism exhibited by the pervasive system means that rapid changes that occur in the system should be reported in the same manner, resulting in producing monitoring data more frequently.

2.5 Conclusion

Pervasive computing is not some obscure idea that is waiting to be implemented in some distant future. It is a technology that is already here and gaining growth, bringing with it a myriad of complicated interactions and perhaps unforeseen consequences in regards to social uses. For computer science though the pervasive computing field brings a whole new set of possibilities and also challenges to overcome.

This chapter presents the idea behind the vision of pervasive computing and the characteristics of the computing system it entails. From the beginning, it is underlined that context-awareness is an inseparable property of pervasive systems. A pervasive computing system integrates with three intertwined environments, namely the users, physical spaces and the computing resources. The principal goal of applications running in this context is to provide useful services to the users, in a transparent manner. This is made possible by using information gathered from the environment and coordinating harmoniously the available devices and services. The understanding of pervasive application scenarios has evolved throughout the years. Lately, more and more application scenarios leverage both local devices situated in user environments and remote computing resources, especially using the Cloud Computing.

The chapter continues on by discussing the characteristics of pervasive environments and applications. These discussions point out new challenges brought by pervasive computing as well as how the existing ones are affected. Among the main challenges, handling dynamism of pervasive environments is undoubtedly one of the most concerning. Pervasive environments are subject to constant evolution. The applications concerned with such environments are required to change according to circumstances and adapt to match the surrounding environment and the needs of users. Without dynamically adaptable applications, pervasive systems cannot offer the flexibility to blend into the real environments.

The final part of this chapter adopts a software engineering point of view in order to take a closer look at the development of pervasive applications. It briefly presents different approaches for tackling the complexity of building applications for the pervasive context. Three categories of software engineering tools are presented as development tools, runtime tools and management tools. While there are many middleware solutions that tackle the challenges of developing and executing applications in pervasive environments, the same cannot be said for the management tools.

The goal of this work is to study the deployment requirements for pervasive applications and provide a solution that manages software deployment in dynamically changing environments. The following chapter introduces the general concepts of software deployment.

Chapter 3

Software Deployment

"Education consists mainly in what we have unlearned."

— Mark Twain

Contents

3.1 Introduction

In order to be in use, any software must be installed and configured. This process is called ***software deployment*** and it consists in the activities that carry software from development into execution.

This chapter introduces the process of software deployment, its terminology and the fundamental concepts. Then, it discusses common issues of software deployment and adjacent domains of software engineering that can be used to address these issues. Finally, the chapter concludes by presenting different approaches that automates the deployment process and comparing them against defined evaluation criteria.

3.1.1 Software Development Life Cycle

Before introducing software deployment, it is important to recognize the broader context in which the deployment is situated: the Software Development Life Cycle (SDLC). Advances in software engineering have radically changed the way software is developed. Previously software was created in two phases, analysis and development. Currently the process of software production has become more methodological and divided into several steps with distinct characteristics. Each step requires specific skill sets, endowed by actors specialized in performing different activities of a software development project. As a result, the software can be seen as a living entity, changing and evolving during its lifetime through a series of activities. SDLC aims to define the tasks, activities and processes required for developing and maintaining software.

The IEEE standard on Software life cycle processes (ISO/IEC 12207-2008) [IEE 2008] defines an exhaustive list of the processes applied during the life cycle of software development. These processes are classified into different groups such as agreement processes, organizational project-enabling processes, project processes and technical processes. The domain of software engineering is mostly interested in technical processes and these are defined as the following.

Requirements analysis aims to define the objectives of the project. It identifies the stakeholders that are involved in the system throughout its life cycle, together with their needs and desires. Then, those are analyzed and reduced into a common set of requirements that expresses intended operation of the system. As in non-software projects, this analysis can be supported by a market research and feasibility study to identify the requirements of stakeholders and if they may be satisfied. Lastly the requirements are transformed into a set of technical requirements that will guide the design of the system.

Design process focuses on creating the skeleton of the software: its architectural design. The system is divided into several elements, and identifies which system require-

ments should be addressed by which element of the system. Each element is specified in terms of expected operations, as well as the relationships between different elements. Architectural design specification acts as a blueprint and eases the future phases of development. It increases the predictability of the project in terms of cost and time.

Implementation consists of the realization of the specified system elements, established during design phase. It consists primarily of programming activity. Resulting software elements must conform to the designated architectural specifications. Individual elements of the software can often be developed in parallel and independently.

Integration process puts system elements together (including software, hardware, other third party systems, etc.) in order to produce a complete system that will satisfy the system design and requirements. This step usually includes the build process, which constructs executable software from the source code. The build process applies operations such as compilation and linking depending on the technology in which the software elements are implemented. At the end of the integration process, the system is ready to be tested as a whole for verifying its quality.

Testing processes are performed transversally to the other phases of the SDLC for verifying and assessing the produced software. Different types of tests validate the system for compliance with the design specifications and requirements. They usually define criteria for assessing the system for delivery. ***Unit tests*** check if the implementation of each system element performs conforming to the design specification. ***Integration tests*** validate if the assembly of different elements behave according to the expectations. ***User acceptance tests*** verify that the resulting software is suitable for the user.

Installation is the set of activities for bringing the software to the target environment and making necessary configurations for the software to run on existing infrastructures. Depending on the environments targeted by the software project, (personal computers, enterprise servers, etc.), these activities can be included in the life cycle (delivered with on-site installation), or left at the discretion of possible users. In either case, the installation should make sure that the system is running expectedly.

Operation is the nominal functioning phase of the software. It represents the final outcome of the software product, operating in its intended environments. This phase is unstable, as the software may stop working or require changes. These problems must be addressed in parallel with the execution of the software in the maintenance phase.

Maintenance process aims to keep the software in a state of optimal performance ***after*** its delivery. Software in operation is subject to malfunctions and changes. This may be due to an error in the development phase (e.g. bug), or a missing feature.

Once the problem is reported, a maintenance team will have to correct the problem, propose a fix and update the application.

Disposal process ends the existence of a software system. It terminates the active support by the operation and maintenance processes, deactivates and removes the product from the target environments. It should leave the environments in an acceptable condition, in accordance with predefined requirements and agreements.

Important point to note is that the entire IEEE standard document does not define or refer to the notion of ***deployment***. Nevertheless, installation, operation, maintenance and disposal processes are described as occurring at the target environment where the software operates. The deployment process presented in this chapter corresponds vaguely to these activities. These technical lifecycle processes outline a linear development for the software project, where each process succeeds the previous one. It is equally acknowledged that software producer organizations are free to make customizations and adjustments to those processes and the way they are applied. The following section presents several well-known development models that propose organization principles on how development processes are applied.

3.1.2 Development Process Models

In spite of the standardization efforts, the definition and coordination of software development activities depend largely on the organization that creates the software. The IEEE standard recognizes this fact and allows customization of their processes definitions. It also recognizes that, above all, software development is a project management challenge. Its foremost problem is to find an efficient way to organize a group of people to create and maintain a reliable, high quality software product, based on customer requirements, such as required features, cost and time constraints. For this reason solutions and practices for developing software depend, to a great extent, on the structure of the organization that develops the software. As expressed by Conway's Law [Conway 1968], the quality of the software is correlated with the quality of the organization structure producing it.

A software development process model describes the activities performed at each stage of a software development project. These models also contain methods, principles and best practices for streamlining the development process. Throughout the years various models have been proposed, bringing adjustments to the development process. New development models allow developers to use the potential of latest paradigms in software engineering and consequently respond to requirements of the software industry. This section describes well-known development process models, comparing the deployment activities considered within.

a. Waterfall Model

Waterfall model is the most basic and oldest of all development processes that formalize the steps of software development life cycle. It involves successive application of development phases as requirements analysis, design, implementation, test, installation and maintenance (Figure 3.1). A phase starts once the previous one is finished. The main convenience of the waterfall model is that it is easy to understand. It lets inexperienced developers to work according to a well-defined, rigid structure. The project requirements are fixed early in the project lifetime, so they are well known upfront by developers and stakeholders. However, the waterfall model is inapt for many of the software projects because of its inflexible structure. It is difficult to respond when the implementation encounters a problem or some of the requirements change at the course of the project. Additionally, passing a lot of time analyzing requirements that are susceptible to change slows down the software creation.

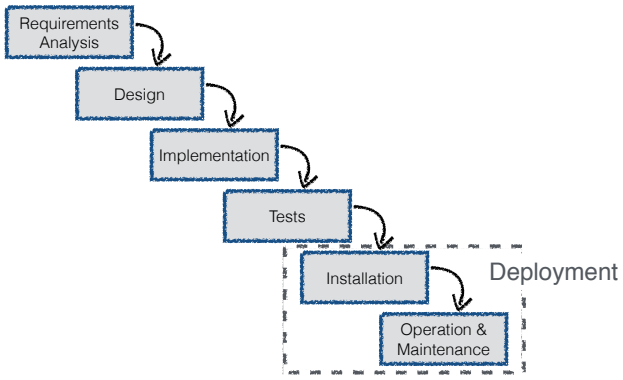


Figure 3.1: Waterfall Model

In the waterfall model, the deployment process is not explicitly described. The software product is delivered once it is entirely developed and tested, meaning that the deployment step happens at the end of the project. It corresponds to the installation, operation and maintenance activities.

b. Iterative Development Model

Iterative development model aims to revise and improve the software product by applying multiple development cycles until it is decided that the software satisfies its requirements (Figure 3.2). Each cycle involves the same sequence of steps as the Waterfall model. Development is done iteratively until the software product is perfected and ready to be released. This way important functions with higher risk factors are developed early in the project, delivered to the customers and receive more feedback for new iterations. Because each release delivers an operational product to the customers, the initial delivery time is reduced. With frequent releases, the development team can react to changing requirements, and adjust the software accordingly for the upcoming release.

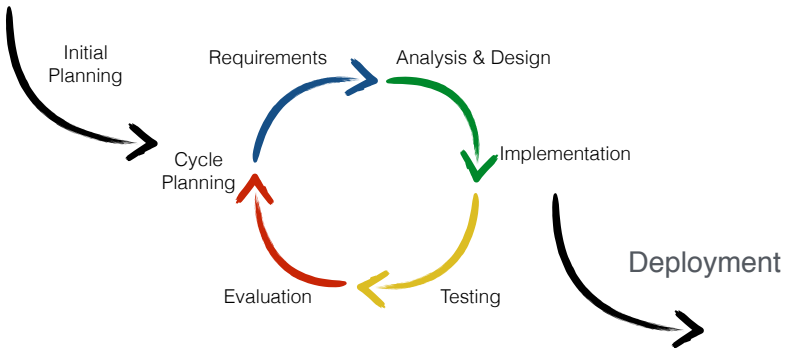


Figure 3.2: Iterative Development Model

In iterative development model the deployment occurs only at the end of certain cycles, when a release is decided to be delivered to the customer. Compared to the waterfall model, the product is deployed much more frequently. Even so, details of the deployment process are not explained in the model.

c. Agile Methods

Agile software development is a group of methods that are based on incremental and iterative development. It combines aforementioned iterative development cycle with incremental build model. Incremental build model proceeds by dividing the system into parts that implement and offer required functions. Then the efforts of development teams are allocated over the set of parts with high priority. Contrary to the monolithic approach where all the different parts are assembled at product release, in incremental development each part is constantly integrated to the whole system as soon as it is completed. Figure 3.3 illustrates this dual functioning[1].

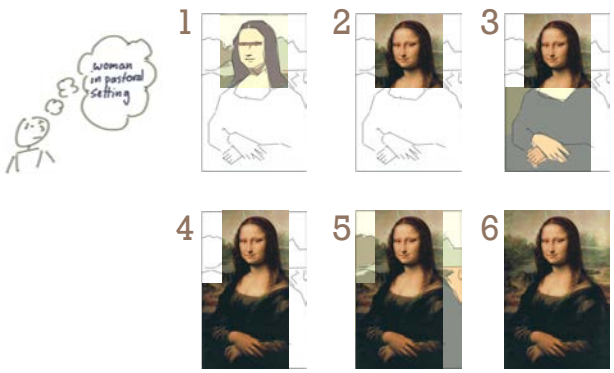


Figure 3.3: Incremental and Iterative Development

Agile manifesto[2] is initiated on the recognition that over planning and over formal-

[1] Original illustrations created by Jeff Patton: http://www.agileproductdesign.com/jeff_patton.html
[2] Agile manifesto: http://agilemanifesto.org

izing hinders the development process, which as a result delays the release of working software products. In order to prevent this, agile software development promotes evolutionary development approach with shorter iteration times. Instead of passing time on planning how to make big design decisions, agile methodology encourages working with the customers closely for understanding their needs and rapidly reacting to the changes. The goal of the development team is to deliver early versions of working software products to the customers, and keep the software in working condition. The deployment gains importance because each development iteration is likely to finish in working software.

In agile development working software is promoted over exhaustive planning and comprehensive documentation. This provokes many critics, arguing that agile methods lack the discipline for developing large-scale software. In [Boehm 2003], Boehm and Turner discuss the dichotomy between discipline and agility. They argue that discipline without agility leads to bureaucracy and slowness, whereas agility without discipline leads to uncontrolled and insignificant enthusiasm. They present agility as a value that augments discipline, for being inventive and adaptive. In fact, compared to more structured, plan-driven development methodologies, agile development requires experienced development teams to adapt to changing requirements and conditions. Development of modern applications that are both critical and dynamic would need to reconcile between agility and discipline.

There are different methods that organize development teams for applying agile principles. Some of the well-known examples include eXtreme programming, Lean Software Development, Scrum and Kanban. Each one of these methods concentrates on different parts of the SDLC. For example, eXtreme programming organizes developers to be more productive in developing high-quality software. It includes practices like pair programming, peer review, extensive testing and minimal documentation. However, it does not describe any constraints or guidelines about the deployment. Scrum provides a framework for organizing requirements definition, development cycles and team meetings. The scrum process relies on the concept of ***sprint***, a two to four week effort focused on developing, testing and deploying a specific functionality. At the beginning of each sprint, the customer can intervene and reprioritize and change requirements of the project. Kanban is another agile methodology that aims to establish a workflow for continuously improving the working product. It is based on a ***board*** on which visually represents the state of feature development. This allows tracking and getting feedback from the advancement of specific tasks and the overall project.

3.1.3 Summary

In addition to the ones presented in this section, there are many other software development methodologies, such as Spiral, V-model and Y-model. In [Larman 2003] authors track down the origins of incremental and iterative software development. They point out a historical shift from strict development models such as waterfall, to iterative and

evolutionary methods, which eventually gave rise to the agile methodology. This shift can be seen as an indicator of the acceleration of software development speed. Despite the known advantages of rigorous documentation and planning in other engineering domains, software development took a turn for rapidity, and frequent release. For instance, a project that applies waterfall method for software development delivers the resulting product at the end of the project, typically after several months of development. Recent agile methods, on the other hand, encourage releasing a version of the product frequently, several times a week, sometimes even at each modification of the source code. In order to cope with this increasing workload, developers are increasingly using more tools to automate their tasks for programming, building, packaging and releasing software.

While software development methods keep accelerating their pace and shortening their iteration cycle, the deployment process needs to keep up with the need for frequent software delivery.

While software development methods keep accelerating and producing software at a higher pace; there is an increasing need for maintaining this flow until the operation, delivering the newly released software to the customers as soon as possible. The research practices in the software deployment process try to answer this problematic. They study activities and models for providing tools that automate and streamline the deployment process.

3.2 Software Deployment

In the previous section the software development life cycle and different methodologies for development are presented. It is shown that recently, software development is seen not as a one-time procedure, but an iterative process that improves the software product and evolves it against changing requirements. This section focuses on the process of software deployment, which is the main subject of this chapter.

3.2.1 Two Faces of Evolution

Software development life cycle paints a vision of software that evolves through its lifetime. The goal of software evolution is to prevent exponential growth of software complexity, in a time computing environments continue to evolve. This evolution leads to consecutive cycles of software design, development and maintenance that continue throughout the lifespan of the software. Such iterative cycles are more and more included in software development processes. As it is discussed in the previous section, agile software development methodologies are built on evolutionary view of software.

New Oxford American Dictionary defines the term ***evolution*** as 'the gradual development of something, especially from a simple to a more complex form' [Stevenson 2010]. Even though evolution is most commonly employed in biology, it is a broader concept that implies changes over time in the characteristics, attributes or properties of an entity or a system [Lehman 1980]. Lehman et al. studied the 'software evolution' to explain the tendency of software programs to steadily increase in size and complexity, becoming harder to adapt. They later described two working areas around this concept, first for understanding the causes, processes and effects of this evolution and second for developing software engineering activities (design, maintenance, refactoring, etc.) to manage effects of it. Lehman et al. classify these modern software programs as ***evolving-type programs***, as opposed to ***specification-type programs*** that are not subject to changes. Evolving programs must be adapted to match any changes in the real world that affect whether the program satisfies its stakeholders' objectives. Since the requirements change constantly, the program must be adapted to continue its correct operation, conforming to its operating environment and stakeholders' requirements. Consequently, unless nothing is done to counter its effects, the software will become more and more complex and unpredictable.

The problem of managing software evolution is addressed at different stages of SDLC. As mentioned previously, agile development methods address the problem of evolution by applying successive development cycles, for changing and improving the delivered product. However, it is also necessary to deliver these changes to the customers through new versions of executing software. On that account, there are two faces of how the evolution is managed, first at development time by managing different versions of developed software, and the second at execution time by delivering the changes to the execution.

The software evolution is handled at two levels; during development by managing the changes on the developed software and during execution, by managing the executing software.

For managing the evolution during development, changes brought to the software should be under control. ***Software Configuration Management*** is a discipline that allows supervising the changes in the software, which in turn serves to control the evolution of the software. The software deployment was initially regarded as a simple extension of configuration management and was not considered a respectable subject of study. Deployment tools were built ad hoc, in the form of scripts that install the software via low-level actions. As the complexity of computing environments increases, the deployment process became an important process of the SDLC. Thus, the transition of the software from development to the execution is covered by the deployment process (Figure 3.4).

This chapter is dedicated to the software deployment. It presents concepts and different approaches and discusses how this restrictive vision of deployment is evolved towards

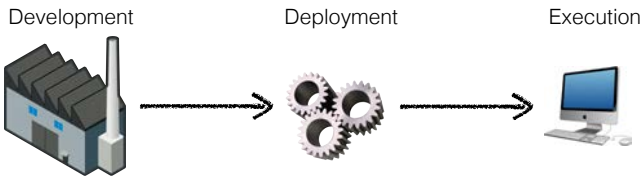


Figure 3.4: Software Deployment

a critical domain which manages software evolution at execution. The following section aims to define the notion of software deployment in general and the concepts used in this domain.

3.2.2 Definitions

Software deployment is a discipline that manages the evolution of a software product after it has been developed. This section refines the general vision of deployment by looking at how it is defined by different researchers. There are different understandings of software deployment according to the domain of interest of the author. Fortunately their definitions do not diverge radically, but emphasize different aspects. For the start, Szypersky confronts the problem of deployment in the context of component-based systems [Szyperski 2003]:

> "Deployment is the process of readying such a component for installation in a specific environment. The degrees of deployment freedom are typically captured in deployment descriptors, where deployment corresponds to filling in parameters of a deployment descriptor."

This definition underlines the importance of componentization in software deployment and introduces the concept of deployment descriptor. For Szypersky, the deployment is installation and configuration of components in an environment. The deployment descriptor can include parameters that serve to configure the components selected with a varying degree of liberty. Object Management Group's Deployment and Configuration of Component-based Distributed Applications Specification [Object Management Group 2006b] (OMG D&C) is a widely admitted reference in software deployment, which describes entities and actors that are involved in the process of deployment. OMG D&C defines the deployment as the following:

> "Deployment is defined as the processes between acquisition of software and execution of software. [...] In order to instantiate, or deploy, a component-based application, instances of each subcomponent must first be created, then interconnected and configured."

This description is conform to the process view of deployment and sees the deployment of a component-based application as the instantiation, configuration and interconnection

of constituting components. Carzinga et al. propose a more general definition of deployment [Carzaniga 1997]:

> "Informally, the term software deployment refers to all the activities that make a software system available for use. [...] The delivery, assembly and management at a site of the resources necessary to use a version of a software system."

In this definition Carzinga et al. acknowledge that the deployment is a set of activities but adds the notion of management to the mix, alongside with delivery and assembly. Moreover it emphasizes that those activities are applied on resources on a site, in order to set up a particular version of a software system among possible others. Lastly Hall et al. details the activities of the same vision in [Hall 1999]:

> "Software deployment is actually a collection of interrelated activities that form the software deployment life cycle. The software deployment life cycle, as we have defined it, is an evolving definition that consists of the following processes: release, retire, install, activate, deactivate, reconfigure, update, adapt, and remove."

Above definitions converge towards a common understanding of the notion of software deployment. The deployment is a process that carries the software product from development to the execution. It consists of various activities which form a lifecycle of the software system.

Definition 3: Software Deployment

Software deployment is the process between the production and the execution of software systems, which involves a set of correlated activities that consists of making configurations and bringing the software to its desired execution state. This process can continue along the lifetime of the software system in order to bring it to a new state via reconfigurations and updates.

3.2.3 Concepts

Following three sections aim to establish a common understanding of software deployment. It presents roles, entities and activities involved in the deployment process. This section, the concepts of deployment, introduces common terms used in most of the deployment systems. Along the introduction of these terms, illustrations enhance and refine the previously mentioned vision of deployment for a more precise description.

a. Component

In [Szyperski 2003], a ***component*** is defined to be a unit of composition with contractually specified interfaces and explicit context dependencies. A component defines its behavior in terms of provided and required interfaces. Deployment plays a central role in Szyperski's definition of component. The first property of components is to be a unit of deployment that is executable in an ***execution environment*** context. Moreover components are a unit of versioning and replacement that is to encapsulate the state they represent. In order to deploy a component it must be instantiated, supplied with instances of components on which it depends and is configured.

b. Assembly – Application

An ***assembly*** is a set of interconnected components. It can itself be viewed as a component made up of subcomponents, and offering and requiring interfaces. The required interfaces of the components in an assembly may be satisfied either by other components in the assembly or be required from the environment in which the assembly is deployed. An ***application*** is simply an assembly of components that are related to each other in order to perform some function. Similarly, in [Carzaniga 1997] a ***software system*** is defined as a coherent collection of artifacts, such as executable, source code, data files and documentation, that are needed at a site to offer some functionality to end users. The Figure 3.5 shows a component-based software is released as an assembly.

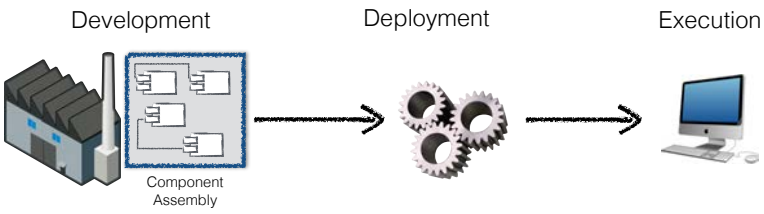


Figure 3.5: Software Deployment – Componentization

c. Deployment Descriptor

A ***deployment descriptor*** captures and describes the artifacts, their configuration parameters, their requirements and relationships, and deployment instructions. It serves to transmit a request of deployment. For example, the deployment descriptor for an application would consist of the components that are included, their relationships, configurations, executable files for those components and specific actions to be taken during the deployment. As shown in figure 3.6, the deployment request is transmitted in the deployment descriptor.

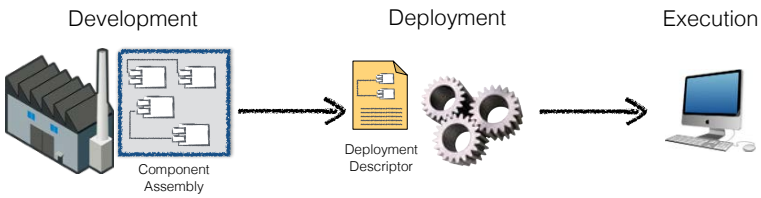


Figure 3.6: Software Deployment – Deployment Descriptor

d. Resource

A ***resource*** is anything needed to enable the use of a software system at a site (both hardware, software and system artifacts). Examples include IP port numbers, memory, disk space and other systems. Some resources may be shared, while others can be used by one system at a time.

e. Site – Target Environment

A ***site*** refers to a single computer that hosts resources. It is part of a network of computers that are administered identically. In OMG D&C [Object Management Group 2006b], the target environment is termed a domain and is comprised of nodes (computers), interconnects (network connections) and bridges (routes between interconnects). As previously mentioned, components are required to execute within a controlled environment known as the ***execution environment*** or the ***container***. At the execution side, the deployment is performed on multiple sites, as shown in Figure 3.7. Each one of these sites constitutes a software system, accommodating a number of resources.

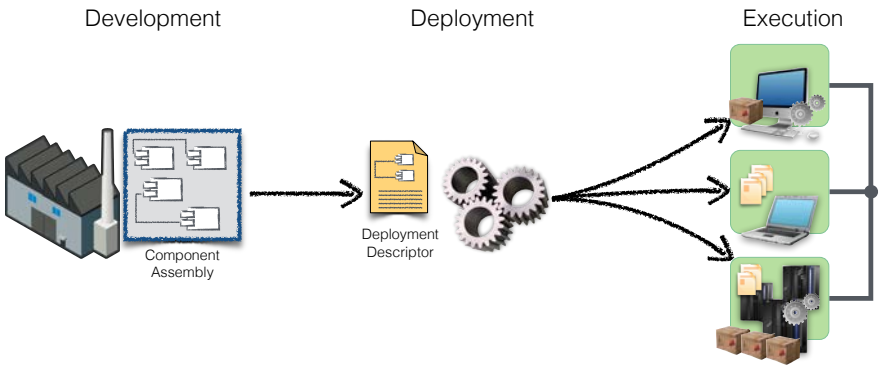


Figure 3.7: Software Deployment – Target Environments

f. Version

A ***version*** of a component refers to time ordered revisions of a component or an application and to platform-specific and/or functional variants [Carzaniga 1997]. Version control systems (VCS) controls, registers and attributes a version to every change made to the documents under its control. This way any change made on development artifacts

are documented, traceable and reversible. As opposed to repository managers, which mostly version deployable artifacts; these systems mostly aim to track the changes made on source documents. Tools like SVN, Git or Mercurial are among most commonly used VCSs.

g. Repository

Deployable artifacts produced from these source documents, are versioned in artifact repositories. A ***repository*** contains the artifacts to be deployed. The repository manager stores and organizes deployable artifacts and meta information about these artifacts. Repository managers are capable of archiving multiple versions of an artifact and analyzing them according to policies indicating product quality such as dependability or performance. They also allow publicly sharing artifact binaries with members of the development team or third-party collaborators. The repository may be located on a central site, which may or not be part of the target platform; or it may be distributed on the sites of the target platform. Regardless of the physical setup, the important point is the logical distinction between the repository and the target. Finally, the Figure 3.8 illustrates how multiple versions of components are handled by the repository and used by the deployment process.

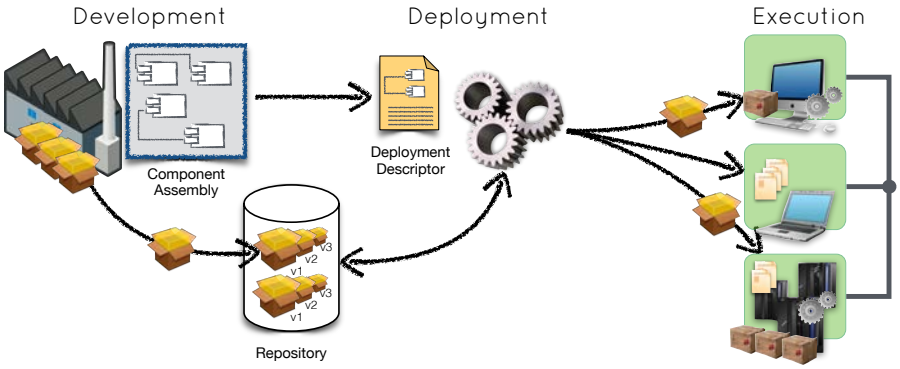


Figure 3.8: Software Deployment (Repository)

3.2.4 Deployment Activities

The process definition of the deployment concept entails the existence of distinct activities. Different studies referenced above, and some others [Dearle 2007, Liu 2006] tried to identify the activities that cover the deployment process as a whole.

Szyperski focuses on the activities for the deployment of particular components. Szyperski identifies four activities for component deployment as acquisition, deployment, installation and loading. The ***acquisition*** refers to obtaining the software component to be deployed. The ***deployment*** readies the component for installation in a specific environment by configuring the parameters. Then the ***installation*** makes the component

available on a particular host of an environment. And lastly ***loading*** enables installed component in a particular runtime context.

Although activities identified by Szyperski are valid for deployment of components, they present a view of a one-time deployment. However, previous sections showed that deployment is an ongoing process that manages the evolution of deployed software. Carzinga et al. propose a more complete view, including activities such as update and adaptation that are intended for evolving the deployed software. Figure 3.9 illustrates these activities. Notice that release and de-release activities involve decisions of development, while other activities occur at the target environment of the software.

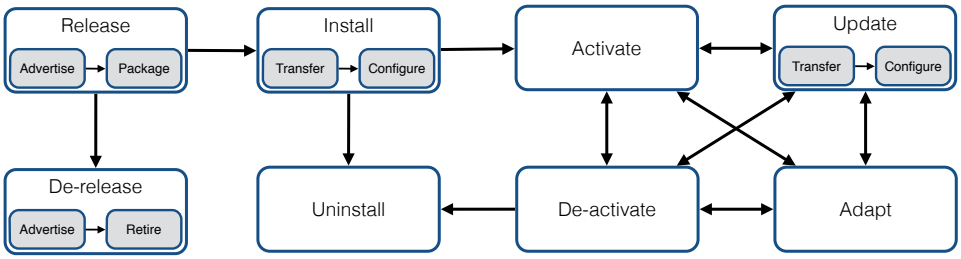


Figure 3.9: Software Deployment Activities

Release is the interface between developers and the actors in the remainder of the software life cycle. At this point the software is assembled into packages containing sufficient metadata to describe the resources on which it depends. These packages become thus the units of deployment. Released packages can be registered on a repository, which will attribute the package with a version, archive it and make the software eligible for access.

Installation is the activity that covers the initial insertion of a system into the consumer site. Usually, it is the most complex of the deployment activities because it deals with the proper assembly of all the resources needed to use a system. It refers to two distinct activities; transfer and configuration.

Activation is the process of starting the software executing or putting in place triggers that will execute the software at an appropriate time. This is sometimes achieved by using graphical interfaces or with scripts or daemon processes.

De-Activation is the opposite of activation, and refers to the activity of shutting down any executing components of an installed system. In general de-activation is required for other activities such as update.

Update is the process of changing a piece of installed software usually triggered by the release of a new version by the developers. Update is a special case of installation but may require installed software to be deactivated prior to update and reactivated after reconfiguration.

Adaptation activity involves modifying a software system that has been previously installed. Adaptation differs from update in that the update activity is initiated by remote stimuli, such as a software producer releasing an update, whereas adaptations are initiated by local stimuli, such as a change in the environment of the consumer site. An adaptation activity may be initiated to take corrective action to maintain the operational correctness of the deployed software system.

Uninstallation activity happens when a software system is no longer required at a given consumer site and can be removed. It presumes the system is already deactivated. The uninstallation activity possibly involves some reconfiguration of other systems in addition to the removal of the uninstalled systems artifacts.

De-release withdraws the system as it is judged obsolete by the producer. As with uninstallation, care must be taken to ensure that the withdrawal will not cause difficulties. This requires the withdrawal to be advertised to all known consumers of the system.

3.2.5 Deployment Roles

Software deployment process happens between the production and execution of software. Therefore two natural roles can be identified; one party that produces the software, the ***producer***, and the other one who consumes and executes it, the ***consumer*** [Carzaniga 1997, Szyperski 2003]. The ***producer*** is basically the role in charge of developing and releasing the software. It is situated at the development side of the software life cycle. Within the organization of the producer role, there are some internal roles that can be relevant for deployment, these are:

Specifier creates the specification of the software to develop. It is a role involved primarily in the design phase, creating the system architecture, including component definitions and relationships. In many organizations the specifier is a senior developer or an architect.

Developer creates the implementations of the specifications. Developers program and produce source code for the implementing components.

Assembler decides on component configurations and interconnections that constitute the software. Those are majorly defined in the architecture specification. The Assemblers job is to choose which implementations will concretely compose the software product.

Packager produces one or more packages that wrap software elements. Granularity of these packages change depending on the technology and the design decisions. Software products may be packaged in an archive file or in a directory that contains all

necessary artifacts. Otherwise, each artifact may be packaged separately and assembled at deployment time.

In parallel to these roles, the ***Repository Administrator*** is in charge of maintaining the repository manager for storing ***packaged*** artifacts for later delivery to consumer sites. Repository administrators can be software producers, consumers or third parties who assemble artifacts from different producers.

The ***consumer*** is the general term for the party who receives and executes the software. It is the consumer-side where almost all of the deployment activities happen. The consumer oversees the functioning of the physical machines of hosts, the execution environments and the software that executes on top of those. The consumer can be refined into these internal roles:

Infrastructure Administrator who operates the physical infrastructures of deployment sites, and in charge of guaranteeing physical resources required by deployed software.

Execution Environment Administrator (or **Platform Operator**) operates one or more execution environments that are found on the deployment sites. The platform operator is in charge of providing software resources required by the software deployed on the execution environment, such as applications.

Deployer is in charge of applying deployment activities (installation, activation, deactivation, update, adaptation and uninstallation) according to deployment requests. In OMG D&C [Object Management Group 2006b] deployer roles are refined as the ***Planner***, who creates a deployment plan describing the actions to be taken for the deployment process; and the ***Executor***, who executes these actions.

3.3 Issues on Software Deployment

Concepts, activities and roles presented in the previous sections establish a basic terminology for studying the software deployment process. This section raises a set of issues and problems that are commonly addressed by current practices of software deployment.

3.3.1 Managing Dynamic Evolution

The evolution of software systems is both natural and inevitable. The evolution is due to the changes in both the software system and its execution environment. Some of these changes can be planned by involved actors, and then executed on the system as such. For example, installation of a new hardware component, such as a network interface, can induce the need for an update of the corresponding driver. A newer version of that driver

can make use of the new component, improving performance and security. Similarly, adding a new feature in an application may require applying deployment actions on different software systems on which the application depends. Nevertheless, in most of the cases, changes are involuntary and unpredictable. A hardware failure (i.e. storage disk failures) or an error of software component (i.e. software bugs) can cause some resources to become unavailable. In such cases the software system may fail, or it can be adapted and reconfigured to function with actual condition of resources.

Emergence of new computing domains, such as Cloud Computing and Pervasive Computing, increases the need for applying deployment activities without disrupting the services provided by the system. Software systems needed for these new domains require to function with high availability while resources are dynamic and volatile. In pervasive systems for instance, communication and integration with devices that are present in an environment is problematic, as these devices can appear and disappear dynamically, without notice. However, this should not interrupt running applications, on the contrary, applications running on a pervasive system should use these changes to their advantage, in order to optimize their behavior.

Managing dynamic evolution is challenging for the deployment process as well as for the execution environment. The ability to apply deployment activities without disrupting the whole system depends on the capabilities supported by the execution environment. Yet, some of the changes brought by deployment activities can occur at runtime, while others may require restarting the system, for the changes to take effect. Consequently, the deployment process should automatically react to the dynamic changes and function with a minimum of human intervention.

3.3.2 Maintaining Metadata Throughout the Life Cycle

As in any design and development process, software producers make decisions throughout the development life cycle. These decisions may include choice of using a programming language, a container, or a software library over another; the description of components and modules, organization and versioning of the source code, structure of other artifacts that are necessary for execution, etc. Such metadata may affect the resulting software product as much as the source code itself. Although developers and software producers in general have valid arguments on why and how they made their decisions, those choices are loosely documented, if at all. Eventually, the information about these design decisions are lost.

One of the ways to overcome this difficulty is to manage every development artifact, source code and design document, using a configuration management system . Along with source and artifact repositories, a configuration management system can be extended to include a metadata repository for storing design decisions. Indeed, a new class of applications called Application Lifecycle Management (ALM) is emerging to propose such

integrated solutions. An ALM goes beyond simple configuration management, by automating workflows and processes performed in the SDLC. It helps creating, assigning and tracking tasks; sharing information between team members and documenting all the inputs made to the software. Using such systems design decisions become visible and their correspondence with functional artifacts are tracked.

Once the software is to be built, released and deployed into execution environment, it is particularly difficult to keep the links between the development artifacts and the actual software at execution. Preserving a direct correspondence between development and execution would help to maintain the software, whether to correct bugs, seek security threats or apply updates. It is thus crucial to maintain the correspondence between development artifacts and the deployed system. The deployment process should confirm which version of which artifact is deployed and effective at runtime, as well as be aware of subsequent changes during the execution.

3.3.3 Managing Heterogeneous Environments

As discussed in section 3.2.5, software life cycle involves many actors comprising end users, software producers and other stakeholders such as platform operators. In modern computing environments it seems impossible to impose a particular configuration of an environment on all the sites that the software is expected to run. For example, the developer team producing the software may work on the Windows platform, while the resulting source code is compiled and integrated on a Linux Server and than at last run on the Linux desktop environment. Additionally, a software product is rarely developed for only a single platform; usually it is destined to run on multiple target environments that have different properties and resources.

Thanks to distributed, large-scale networks, heterogeneous hardware platforms such as servers, personal workstations and mobile devices more and more coexist in same computing infrastructures. These hardware platforms can host software systems that are connected to each other through standard communication protocols. Even if these environments share a major number of properties (i.e. hardware configuration, operating system, ...), each system is unique with different software dispositions.

Heterogeneous target environments challenge deployment in various fronts. First of all, the software that supports deployment has to function in every target platform. This means that the deployment software must recognize every type of resource in those environments and know how to deploy (install, reconfigure, uninstall) resources. Moreover, deployment procedures of similar resources can be different in different platforms. The software that automates the deployment should be generalized enough for handling similar resources, but also specialized enough for managing heterogeneity of platforms. Often, it is necessary to extend the deployment to able to handle new types of resources and software systems.

3.3.4 Managing Dependencies

Reutilization and modular design has become de-facto principles of software development. Recently, with the proliferation of the Internet and web technologies, any non-trivial software system consists of multiple modules with dependencies to applications or external libraries. Installing an application composed of multiple components requires installing all components and ensuring that they can function properly, i.e. all of their dependencies are satisfied.

There are different types of dependencies that software systems are subject to. First, in a software system, components constituting the system may have dependencies between each other. Resolving such dependencies and managing them at execution time requires efforts on both development and the deployment process. Components may manifest dependencies during different life cycle stages. For instance, a component may not have any dependencies for its installation, but may need the functionality proposed by another for its correct execution. Secondly, software systems may have dependencies to the resources or functionalities provided by the target execution environment. An application requiring access to a file in the filesystem is an example to this use case. Thirdly, there may be dependencies between software systems. For example, a flight booking application would need to access to another application, possibly managing a database, for querying available seats and the prices. In this respect, the deployment process should ensure, before deploying the booking application, that the database application is available on site or remotely; and configure both systems for guaranteeing the communication between the two.

The promise of modular programming is to separate the concerns such that different modules perform discrete functions. Separated into modules, the code base of software is easier to maintain, that is to develop, debug and update. It is also easier to reuse these modules in other software products. However, in a modular system interactions between modules pose several problems. Module dependencies significantly increase the complexity of the assembly and the deployment process. The deployment process should resolve dependencies of each module, assemble needed versions of those, deploy them separately and ensure that they are linked together to form the expected software.

3.3.5 Planning and Coordinating Deployment Activities

A significant concern is regarding the planning and coordination of the deployment process. Deployment planning is the operation that decides the actions to be taken during the deployment process. The plan is calculated with the given deployment descriptor and the state of the resources of the target environment on which the software will be executed. Once the deployment plan is constructed, the deployment process should coordinate the decided actions, possibly by targeting distributed sites, in order to successfully execute the deployment process.

The calculated deployment plan is the outline of the actions to be applied during the deployment process. It should answer several questions regarding how the deployment will proceed:

- ***What:*** What are the artifacts (components, files, etc.) that will be brought to the target environment and installed? What are the resources that are already on the target sites but needs reconfiguring or adapting?
- ***Where:*** Where the software system and its components will be placed? Which component will be placed on which target site?
- ***When:*** When will the deployment actions occur? Is there a need for synchronizing different actions, or can any two tasks be executed at the same time?

Calculating answers to those questions is challenging, especially when there is a large number of component and target site combination. In fact, without any indications set by the deployment descriptor, this ***component placement problem*** is a special case of quadratic assignment problem, which is NP-hard [Garey 1979]. For this reason solutions for a deployment plan require a degree of guidance to restrict the possibilities. This information can be acquired as policies described in the deployment descriptor, deployable artifact or target site description.

Once the deployment plan is decided on, it is to the deployment system to coordinate the actions on possibly multiple sites for accomplishing the process. In general, most deployment activities take place at the consumer site. They make use of system resources and often require exclusive access to system components. Also, a deployment action might introduce conflicts with installed or running software.

3.3.6 Ensuring Security

The capabilities of a deployment system are in vain if they compromise the security of the deployed software and the target site. In an enterprise environment, computer security is a prime concern, especially when it is about the management of distributed network of machines. There are three aspects of computer security that are critical with respect to software deployment: authorization, privacy and integrity.

Deployment actions require usually access to critical resources of the system. Reliable authentication procedures must be in place to ensure that deployment processes are started and conducted only by authorized actors. The organizations are rightfully concerned about the privacy of the information they transmit into the network. In the case of deployment, transferred deployment artifacts (for instance, database files, data structures, etc.) may contain sensible information that the organization wants to make it private. Providing this level of privacy may require several things. First of all, deployment process must make sure that the connection between the two parties of the file transfer

is authentic. Secondly, signatures and encryption can be necessary to guarantee both the authenticity and privacy of the artifacts content. These prevent a third-party to look or change the contents of the transferred artifact, which is valid for a file as well as for an executable component.

Even if the transfer of software is carried out in a secure way, there might still be security concerns related to the installation of software in the final target environment. In particular, it is important to guarantee the integrity of the organization's data against the execution of malicious or incorrect procedures that may cause corruption or loss of data during installation or update.

3.4 Software Deployment and Other Research Fields

Like most of the domains of software engineering software deployment is not an isolated research domain. As it is situated between the development and execution, the techniques and methods employed for deploying software is highly influenced by the advances in development and execution platforms. Therefore it is impossible to study software deployment without understanding these adjacent domains. Different domains of software engineering have addressed issues discussed above, and these domains have contributed on how the deployment process is conducted. This section presents these domains and how they contribute to the deployment process.

3.4.1 Software Architectures

Software architecture is a design artifact that records and justifies important design decisions of a software system. It abstracts information on different views of the software system, notably regarding its structure and evolution. The architecture is a description of the expected system, including components, relationships between them, constraints on their execution, etc. But also it is a prescription of how the system can evolve; principles, restrictions and guidelines that may be presented as architectural patterns and styles. While historically software architectures are design artifacts created in development, their usage increasingly shifts to the heart of execution [Baresi 2010].

The research community studies the usage of specific languages for describing and manipulating software architectures, named Architecture Description Languages (ADL). ADLs intend to represent one or more architectural views focusing on a particular concern. An ADL can be designed in different forms; as informal (e.g. use of schemas), as semi-formal (e.g. UML) or as formal. Architecture described using a formal ADL can be interpreted by a machine for evaluating and automating certain aspects, such as design, deployment and execution [Medvidovic 2000].

In software deployment field, architecture-based deployment is a common term for describing the usage of architectural description as deployment descriptor that guides

the deployment process. Architectural models are adequate for this because they already contain information about the elements contained in the software and their relationships. The architecture of a system can be served as a base model for associating metadata about deployable artifacts. This information can then be leveraged throughout the deployment as discussed in section 3.3.2. As for relationships, they usually represent a kind of ***use*** relation between the elements they involve. Therefore, they can be interpreted as dependencies between software elements. The issue about managing dependencies is discussed in section 3.3.4. Dependency information contained in architectural models can also be used in the deployment process.

The software deployment process is about maintaining evolution of the software system at execution time. A more recent class of ADLs addresses this issue by allowing dynamic architectural manipulation. C2, Rapide, Darwin and Weaves are some of the examples for Dynamic ADLs. These provide operations and languages for modifying the architectures by adding, removing and rewiring elements at runtime. However, many of the current ADL's does not cope well with expressing dynamic changes [Medvidovic 1996]. Changing a software specification written with an ADL introduces many problems related to the deployment. Migrating the system into a new architectural specification triggers a set of adaptation and update activities in which components can be created, destroyed, reconfigured while sometimes saving their internal state. In [Dearle 2007], Dearle states that such activities do not only require languages to express these operations, but they also need to be capable of expressing the complex temporal and transactional state space that occur during reconfiguration.

3.4.2 Software Product Lines

A software product line (SPL) is a set of engineering techniques for developing software systems. It favors reuse of artifacts by defining product families that share common features [Bosch 2000]. Like in industrial product lining, in SPLs, software products are divided into groups of closely related products, to offer them separately in different situations. The concept of a product family defines the whole of the configuration space, including points of variability over possible products. A software product can be seen as a particular configuration of reusable artifacts, composed in accordance with a number of constraints and preferences. SPLs aim to improve the time to market, productivity and quality of software products by promoting reusability.

The deployment process is involved in this when the product, meaning the resulting application configuration, is released and delivered to the consumers. Once released, the products can be delivered rapidly using an automated deployment process placed at the end of the product line. Releasing an application from a product family requires deciding on an application configuration. In component-based, modular systems the choice of assembled components can define the application configuration. Whereas in monolithic systems, these configurations are made when the software is built through a customiz-

able build process. For example, tools like Maven and make files allow to define such custom build processes and releasing software for different target environments. These techniques applied in product line practices allow software producers to address heterogeneous target environments discussed in section 3.3.3.

The variability over possible choices in a product family is represented in models called ***reference architecture***. Reference architectures include shared architecture of a product family and additional information for variable features. When constructing an application, the SPL is confronted with the problem of resolving dependencies of the expected product configuration (discussed in section 3.3.4). Reference architectures can be refined with choices made on the product release to calculate the effective architecture of a particular application and resolve the dependencies of the application.

Traditional SPL engineering advocated that variation points are bound before the delivery of the software. More recently Dynamic SPLs (DSPL) emerged, where selection and binding of the variation points are realized dynamically at runtime [Hallsteinsen 2008, Bencomo 2010]. These systems use the variability model that is expressed in the reference architecture for adapting the running system. The deployment process of such systems should evaluate the variability model in permanence in order to change chosen variation configurations.

In [Cetina 2008] Cetina et al. presents a discussion interesting from the point of view of deployment process. In this paper authors define the difference between two architectures of DSPLs as connected and disconnected. In the connected DSPL architecture, the configurable product is always coupled with a product line, from which it receives adaptation requests. In a disconnected DSPL architecture the configurable product is more autonomous. It embeds the product line model (reference architecture) and applies adaptations by making decisions based on this model. These approaches indeed show two visions of deployment. First, the deployment is decided remotely and guided by requests sent to target sites. Second, the deployment is decided and conducted essentially on local site.

3.4.3 Self-adaptive Software Systems

Installed systems must evolve to address changes in both the environment in which they operate and the requirements they fulfill. As presented above in software architectures and product lines, the ability to change software systems dynamically is a demanded property for coping with planned and unplanned evolution. Self-adaptive software systems are able to adjust their behavior in response to their perception of the environment and the system itself. Engineering self-adaptive systems pose major challenges. These systems should be aware of the environment, take decisions and be able to change their execution accordingly.

Self-adaptive systems propose the primitives for a deployment that covers the adap-

tation activity. Deployment on a self-adaptive target would decide on the actions to be taken, and coordinate the execution of these actions on the system. In [Oreizy 1999] Oreizy et al. distinguish these two processes in self-adaptive systems as evolution management and adaptation management. On one hand the ***evolution management*** aims to maintain the consistency and integrity of the system over time based on architectural models. On the other hand, changes and observations needed by the evolution management are applied by the ***adaptation management*** (issue 3.3.1). The adaptation management is in charge of detecting the inconsistencies, planning and deploying modifications (issue 3.3.5).

With these operations, a self-adaptive system can be seen as a closed-loop system with feedback from the environment and itself. Autonomic computing [Kephart 2003] proposes the MAPE-K architecture for implementing this adaptation loop, including Monitoring, Analyzing, Planning functions and a shared Knowledge-base. Autonomic managers that interact with the managed system via sensors and actuators implement this architecture.

There are a number of obstacles to overcome for engineering the execution environment for self-adaptive systems. First, a self-adaptive system should be aware of itself and its environment, monitoring the changes and being notified about them. This includes an ***introspectable*** execution environment, meaning that it should provide means for inspecting its architecture. Additionally, information about its environment context should be gathered and modeled within the system. Second, the information about the system should be ***analyzed***, and the self-adaptive system should make ***decisions*** on the actions to take. Many approaches are invented and borrowed from other domains for analysis and decision functions [Salehie 2009]. Policies, rules, QoS definitions and artificial intelligence techniques are some of the approaches most commonly used. Lastly, decided actions should be ***effectuated*** through an infrastructure that allows managing the system and making changes at runtime.

3.4.4 System Administration

The correct execution of a software system depends on its stability and harmony with its environment. The goal of the system administration is to ensure the stability of execution of computing systems both hardware and software.

System administrators are in charge of supervising the whole system hardware and software. They make sure that the system provides resources needed for the execution of the applications and services. Their goal is to ensure that the computing system is operating with optimal performance and uncompromised security, without exceeding the requirements of maintenance costs. The domain of system administration (also called ***IT administration*** or ***operations***) is decoupled from the development of the software. The software development seeks developing new features, optimizing existing ones and fixing

bugs thus evolving the software system. While the system administration is about trying to keep the system as-is, once it is at good operation, and is interested in evolving the ecosystem in which the software system lives. This may involve tasks such as migrating systems to new environments, running back-up procedures and troubleshooting the errors.

In order to accomplish these tasks, system administrators usually execute one or many activities of the deployment process. However, historically they are used to interact with the systems via low-level tools. They usually run commands via command-line interface, or in some cases use ad-hoc scripts they have written for automating some recurrent tasks. Supporting the tasks of system administrators with well-defined deployment processes would not only automate these tasks but also reduce human errors that occur during deployment activities.

3.4.5 Summary

The previous section discusses the issues encountered in the software deployment process. The problem of software deployment stands out as a collection of intricate issues that involve many research fields in software engineering. This section presented the fields that already address aforementioned issues. An important point to remark is that software deployment has two-way relationships with those domains. Meaning that all of these domains involve and apply deployment processes and they also contribute to the way the deployment is conducted by resolving issues. The table 3.1 shows a summary of tackled issues by these research fields. The following section presents existing efforts, both academic and industrial, that tackle these issues. Some of the important deployment automation solutions are evaluated against criteria that are also presented.

Table 3.1: Software engineering fields responses to issues

Issues	Software Architectures	Software Product Lines	Self-Adaptive Systems	System Administration
Managing Dynamic Evolution			✔	
Maintaining Metadata Throughout the Life Cycle	✔			
Managing Heterogeneous Environments		✔		
Managing Dependencies	✔	✔		
Planning and Coordinating Deployment			✔	✔
Ensuring Security				✔

3.5 Software Deployment Facilities

Automating the software deployment process is the only way to cope with increasing speed of the cycle of development, release and delivery. There are a large variety of tools to help producers and consumers to deploy their software. These tools provide different degrees of automation over deployment activities presented in the subsection 3.2.4. All the same, regarding automation, it is important to recognize that the deployment process is a part of the SDLC, where human participants conduct most of creative tasks in different processes. For example, developers are in charge of producing the creative content (code, configuration files, etc.) although tools of modeling and programming aid them.

Software deployment is about organization of human processes, as much as it is about the tools that help its actual process. Therefore the models used to represent the process and the practices employed during it are as important as the tools themselves. It is useful to define a concept that includes all conceptual and software tools that helps automating the deployment process. The following definition describes this concept of ***software deployment facilities***.

Definition 4: Software Deployment Facilities

Software deployment facilities defines the group of models, processes and tools employed by an organization for handling deployment processes by optimizing and automating its tasks.

3.5.1 Characterization Framework

Before presenting different technologies and academic works that propose software deployment facilities, this subsection is dedicated to the conceptual framework that is suggested by Heimbigner et al. [Heimbigner 1998]. This conceptual framework aims to characterize different capabilities expected from a deployment facility and used by the authors to classify existing technologies. Presenting these capabilities serves for evaluating different deployment facilities presented in this section.

a. Process Coverage

The first characterization criterion is the process coverage, the degree to which a deployment system covers each of the deployment activities of the process. The subsection 3.2.4 describes activities that constitutes the software deployment process. An activity is covered if the deployment system provides full support, meaning that it implements at least a default version of the activity and describes how it can be integrated to the whole process. An activity is partially covered when the deployment system does not provide an implementation but recognizes the existence of the particular activity and provides means to

the user for implementing and integrating the activity to the process. The process coverage criterion evaluates the completeness of the deployment automation solution. Finally an activity is not covered by the deployment system if it does not explicitly recognizes as part of the process.

b. Process Changeability

The second of characterization criterion is the changeability of the deployment process. It is difficult to define and implement a deployment process for every possible use of software product and consumer site. Typically, a particular product can require a special procedure for deployment, or a consumer site may need to run specific test before validating the deployment. This implies that a rigid, non-changing deployment process is not applicable to all possible use cases. Process changeability indicates the ability of the process to be changed and be customized after definition. A changeable deployment system should allow customizing the deployment process per consumer in order to include additional steps to some deployment activity.

c. Interprocess Coordination

A complete deployment process would most possibly include coordination of various deployment activities on different software systems. Additionally these systems can be distributed over different sites and should be synchronized for the sake of the coherence of whole deployment process. For example, updating a component may require, first to deactivate other components that depend on it, then updating the first component and only after that reactivating its dependencies. The interprocess coordination criterion evaluates the deployment system's ability to coordinate activities and synchronize between distributed processes.

d. Site, Product, Policy Abstraction

The final characterization criterion is about how activities are described in the process definition. A deployment activity can be seen as a procedure for controlling execution of actions that manipulate resources on consumer sites. Therefore an activity can be described in terms of the consumer site, the product or components of the product and a set of execution policy constraints.

There are many ways to program deployment activities for implementing a deployment system that automates the deployment process. The straightforward way of describing a deployment activity is to program execution procedures for each combination of product and consumer site with every kind of execution policy. These execution procedures are usually developed with general-purpose scripting languages such as Perl, Python and Ruby. Clearly this can lead to a large number of such scripts and the consequent high cost of their individual development and maintenance. Another way of describing deployment activities is to factor out common information about product, consumer site and policies inside abstract models. Modeling information about these entities

reduces the effort required to define deployment processes, and allows to use the same abstractions in different range of situations. Thus, the deployment procedures themselves become generic, reducing the total number of deployment procedures that must be defined. These generic models can then be parameterized with information specific to the particular deployment process.

The site model, the product model, and the policy model characterize a deployment systems ability to describe information about the deployment activities. The following are more detailed descriptions of these models.

The Site Model The site model is a standardized way of describing or abstracting a consumer site's resources and configuration. A site model for a single computer would contain information such as the machine type, the operating system, the available hardware and software resources.

The site model enables all consumer sites to be treated in the same manner, regardless of their nature. This way all consumer sites can be treated in the same manner, regardless of their particularities. A unified model would provide standard methods to access the site's configuration and to manipulate required resources for performing deployment activities. The deployment system then can ignore differences between consumer-sites. In this respect the site model specifically addresses the issue of heterogeneity discussed in section 3.3.3.

With a site model the deployment activities are greatly simplified, since a deployment system can access the common information from the site model to use in deployment activities. Autoconf and Windows Registry are two examples of simple site models for respectively Linux and Windows platforms. Autoconf is used to produce procedural shell scripts from configurations by dynamically computing the site abstraction. Windows Registry, in contrast, is a passive repository containing the site abstraction.

The Product Model The product model describes the constraints and dependencies of the system to be deployed. The deployment system uses this model to reason about all deployable and deployed products, in order to ensure that the target site is consistent. The product model should include information about the content of the product, such as the set of required files and components, dependency specifications, general information about the producer and documentation. The deployment descriptor defined in 3.2.3 usually contains the product model, or enough information to construct the product model. Constructing the product model can be straightforward for a monolithic system, whereas modular products that will be deployed in distributed environments increase the need for more expressive models.

Throughout the deployment process, the product model is queried by the deployment system for gathering the information needed to execute deployment activities. It is often the case that the product information is integrated into the site informa-

tion once a system is installed. In section 3.2.1, it is discussed that configuration management tools also schematize information about the software products. This integration indicates the link between configuration management and deployment systems.

The Policy Model A deployment policy is a particular way of customizing the execution of a deployment activity. It defines how the standard deployment activity is changed for that particular deployment. The policy model can include information describing aspects such as scheduling deployment requests, preferences, and security control. For example, in case of a modular product, the integrity and compatibility of constituent components should be verified. A strict policy would be to perform these checks beforehand, and consider starting the deployment accordingly. Or a looser policy would start the installation with a minimum of verification and then validate the deployment once all components are in place. Another example of alternative policies for the same activity concerns whether updates should be pushed or pulled. Under both the push and pull policies, the installation activities are essentially the same, differing only in when and how updates are triggered.

Usually different policies are hard-coded within the deployment system and not externalized in policy models. It is difficult to construct a deployment system that can be extended with new policies. Instead of modeling various policies, many deployment facilities choose to provide hooks for developers (either for product developers or consumers) to react to different stages of the standard deployment process.

3.5.2 Evaluation Criteria

Creating abstract models of these aspects is of major importance for automating the deployment process. This is particularly apparent in distributed environments, where heterogeneity and coordination issues should be handled in order to provide a successful deployment environment. Previous section presents the characterization framework proposed by Heimbigner et al. [Heimbigner 1998]. Authors use this framework for evaluating some of industrial solutions. However, their evaluation lacks concrete indicators of capabilities expected from deployment solutions. To serve as evaluation criteria, here a number of indicators are identified and grouped into three categories:

Deployment Platform defines the technology stack on which the consumer sites are constructed. The characteristics of the deployment platform are as important as the deployment system itself. The capabilities of deployment solutions are naturally limited by those of the deployment platform.

- ***Deployment Unit:*** The kind and granularity of the deployment unit.

- ***Modularity:*** Whether the platform provides a modularity layer that allows to load and unload modules.
- ***Site Representation:*** Whether the platform provides a representation of the resources available on the platform and site in general.

Deployment Process defines the characteristics of the process proposed by the deployment facility.

- ***Deployment Activities:*** The set of activities defined by the process for evaluating the process coverage.
- ***Process Hooks:*** Whether and where the process defines places in the process to attach customization policies.
- ***Distributed Coordination:*** Whether the deployment process can be coordinated on multiple distributed sites.

Deployment Description defines the kind of deployment description and the capabilities enabled by it.

- ***Deployment Descriptor:*** The kind of deployment description artifact.
- ***Descriptor Placement:*** Whether there is an independent descriptor artifact or if not, how does the deployment descriptor is kept.
- ***Policy Description:*** Whether the descriptor lets defining custom deployment policies for extending default deployment process.

The rest of this section presents software deployment facilities proposed by industrial products and the research community. They are divided into four categories according to their operation scope. At the end of each category some of the solutions characterizing that category are evaluated against these criteria.

3.5.3 Single Target Deployment

Single target deployment comprises technologies that consider a single machine as their target consumer site. Automating the deployment process is relatively unchallenging, as the deployment system does not need to deal with issues like heterogeneity, planning and coordination on distributed environments. Additionally, automation requirements are less elaborated as the main goal of these systems is to help end-users install applications. Nevertheless, numerous tools proposed in this field have constituted the foundational effort for deployment automation in general. These technologies are studied in three categories.

a. Package Managers

RPM Package Manager (RPM) [Bailey 1997] and dpkg are examples of package managers, widely used low-level deployment tools for Linux and UNIX-like operating systems. These utilities are capable of querying, verifying, installing, uninstalling and updating software packages. They propose command-line interfaces for accessing information about packages and executing deployment actions. A ***package*** is defined as a collection of files, configurations, documentation and metadata such as description and signature. In general, packages are required to be associated with a version, which allows to handle multiple revisions of the same package. Package managers use repositories where packages are stored and indexed. A package repository is a remote database containing metadata about available packages. A local database is also used to register that are changed and created when a package is installed. This can revert the changes and remove an installed package, without breaking existing ones.

For handling the deployment of multiple packages, package managers model applications as a graph of interdependent packages. This brings the problem of managing dependencies of package to be installed. Numerous higher-level tools for software package maintenance exist such as Yellowdog Updater Modified (YUM) and Advanced Packaging Tool (APT). Their automated deployment operations such as retrieving, installing, updating, and uninstaling applications, calculating the tree of dependencies.

Package managers are the most common way of delivering software in general. There are many examples of package managers, each specialized in deploying packages required for the respective technologies. NPM[3] for Node.js, Ruby Gems[4] for Ruby, NuGet[5] for .Net are some examples of package managers specialized per execution environment. NPM, for instance, allows the deployed software to declare scripts that will be called on certain deployment stages such as pre- install, post- install, pre- start, etc. Homebrew[6] is another tool that is specialized for installing Unix-like packages in Mac OS environments. It is based on package descriptions called ***formula***. A formula can cite other packages it depends, resources it needs to download and finally a script (written in Ruby) that applies the installation. This lets Homebrew to download directly the source code of the program and compile it on-site.

b. Application Installers

Compared to the package managers, Application Installers provide an application-centric deployment model. Tools such as Windows Installer and InstallShield handle applications on the basis of features and components. A ***feature*** represents an application functionality that users may or may not decide to install. Features can be installed independently from each other. A ***component*** is the part of an application to be installed which is hidden

[3] Node Packaged Modules: https://www.npmjs.org/
[4] Ruby Gems: https://rubygems.org/
[5] Nuget: https://www.nuget.org/
[6] Homebrew: http://brew.sh/

from the user. Applications installers usually propose a standardized user interface, where the user can choose one or more features for installation. Then the installer determines which components must be installed in order to install that feature. It is up to application developers to decide how to divide their application into features and components.

Application installers also use a local database for tracking which applications require a particular component, which files comprise each component, where each file is installed in the system, and where component sources are located. The deployment process consists of acquisition of features to be deployed, calculation of components to be installed and execution of component installation. The installation phase comprise the execution of predefined scripts of installation. If the installation process fails, a rollback process can revert the changes.

Comparably, IzPack[7] is a tool that applies principles of application installers, which are predominantly in Windows environments, to applications running on the Java technology. IzPack allows developers to create customizable software packages that can be deployed in multiple environments. It lets application developers to specify deployment policies that are conditional on the parameters that differ from one target environment to another.

c. Web-centric Deployers

Web-centric Deployers emerged with the proliferation of Internet, for transferring software in a controlled, secure way. Several technologies support the web-centric deployment model. Java Applets, ActiveX components, Java Web Start [Sun Microsystems 2006a] (a reference implementation of Java Network Launching Protocol (JNLP) standard), .Net ClickOnce[8] and ZeroInstall[9] are such examples of web-centric deployment technologies. The web-centric deployment aims to transparently transfer executable software artifacts from a web server to the computer of the end-user. As security is a major preoccupation in web technologies, unless trusted, applications run in a protective environment, a sandbox, with restricted access to local deployment site resources.

Web-centric deployment techniques can divide software into smaller components. This enables incremental retrieve and update but is prone to the dependency management issue. In order to overcome this problem, web-centric applications are usually packaged independently. Instead of sharing their components, artifacts of each application is downloaded and stored separately. Another functionality proposed by web-centric deployers is the ability to detect missing runtime environments (Java Runtime Environment (JRE) or Common Language Runtime (CLR)) and automatically installing the required runtime. This brings increased transparency for the users, though it is prone to conflicts between

[7] IzPack: http://izpack.org/

[8] MS .Net ClickOnce: http://msdn.microsoft.com/en-us/library/t71a733d(v=vs.80).ASPX

[9] Zero Install: http://0install.net/

runtimes needed by different applications.

To conclude single target deployment, the web-centric model ensures significant automation of the deployment process for single computer machine. The presented mechanisms find their use also for more complex execution environments. In such a case, however, they need to be supported by additional deployment tools to provide full deployment automation. In the following subsection we present execution platforms that enable deployment of independent modules.

Table 3.2: Comparison of single target deployment facilities

Criteria	npm	IzPack	Java Web Start
Deployment Unit	Package, compressed folder with package descriptor about dependencies	Pack, files grouped under a package ID	Resource, set resources such as Jar files, native libraries and system properties
Modularity	Node.js module structure	Standard Java modularity	Standard Java modularity
Site Representation	Installed packages, Key-value properties, environment variables	Key-value properties, environment variables, Windows Registry	Environment variables, operating system, processor architecture, JVM arguments
Deployment Activities	Publish, install, remove, restart, start, stop, update, uninstall, unpublish	Parameter collection, Install (file copy, parse, execute), Uninstall, Reporting	–
Process Hooks	Script hooks on test, start, restart, stop	Listeners before/after install and uninstall	–
Distributed Coordination	No coordination, network connection between client and package registry	No coordination	No coordination, network connection between client and web server
Descriptor File	package.json citing dependencies	Installation description (XML)	Jnlp file (XML)
Descriptor Placement	Contained in packages	Used for creating the installer	Independent from resources
Policy Description	Scripts other than process hooks	Custom actions	–

3.5.4 Modular Execution Platforms

Deployment of traditional applications depends on external deployer facilities, such as the ones presented in the preceding section. A deployment system needs to calculate or extract information on applications such as dependencies, geographical distribution on target sites, availability of required resources on these sites etc.

In a broader view component-based programming is based on the modular design principles for software development. With component-based programming a major effort was made to facilitate the deployment of component-based applications, the objective being to predict the phase of the deployment during the development. That is to say, in component execution platforms, models explicitly provide means to describe components (or

modules) and their dependencies. This section presents three important execution platforms that promote developing modular software by providing execution environments that host the components. More specifically, the emphasis of these discussions are on how the modules or components are described and packaged, how does the deployment descriptor of those is created and in which ways these platforms addressed previously introduced issues.

a. CORBA Component Model

The Common Object Request Broker Architecture (CORBA) is a standard defined by the OMG that enables software components written in different computer languages and executing on multiple computers to work together. Corba Component Model (CCM) [Object Management Group 2006a] is a component-based execution platform of distributed CORBA.

In CCM, a component is defined by an interface and one or more implementations of that interface. A CORBA component is a unit of deployment, that is to say it is the basic element of the deployment. It consists of a ***zip*** archive containing the description of the component files, the implementation binaries and a file to express the properties. An assembly of CCM components is a set of logically interconnected components distributed over multiple machines. During deployment, the assembly will be physically installed on a given configuration machines by establishing connections between components. To describe the assembly, CCM uses a descriptor file (***.cad*** for "Component Assembly Descriptor"). An assembly description is composed of assembly packages that contain the assembly handle and a set of component packages, containing the components involved in assembly.

CORBA runtime provides tools to realize the deployment phase. The deployment activities include transfer, installation, composition, instantiation and configuration of components on targeted runtimes. CCM specifies a number of steps to take during the deployment process: the definition and selection of deployment sites; installation of implementations using the information contained in the descriptor software package, instantiation of components and finally connection of components.

b. EJB

Enterprise JavaBeans (EJB) is a managed, server-side component architecture for modular construction of enterprise Java applications. The EJB specification is one of several Java APIs in the Java EE specification [Sun Microsystems 2013a]. The EJB specification intends to provide a standard way to implement the back-end 'business' code typically found in enterprise applications (as opposed to 'front-end' interface code). Such code addresses the same types of problems, and solutions to these problems are often repeatedly re-implemented by programmers. Enterprise JavaBeans are intended to handle such common concerns as persistence, transactional integrity, and security in a standard way, leaving programmers free to concentrate on the particular problem at hand.

EJB specification defines the installation, activation, deactivation and uninstall beans. However, contrary to what could be expected, the bean is not the unit of deployment that has been used. The deployment of beans or the applications based on beans are defined via an archive file. These archives can contain an XML file playing the role of deployment descriptor. This descriptor typically contains information required for each bean in terms of transactions, security and persistence. Finally, it should be noted that the archive format and content of the deployment descriptor are the main elements of the standard defined by the EJB specification for units of deployment.

EJB specification does not address the problem of coordinated deployment of beans on multiple distributed application servers.

c. OSGi

OSGi is a service platform specification, which delivers an open common architecture for service providers, software developers and equipment vendors to develop, deploy and manage services in a coordinated fashion [OSGi Alliance 2007]. It enables flexible and managed deployment of services, based on a modularization model for Java Runtime Environment (JRE). OSGi defines deployment units, called ***bundle*** that contain compiled Java code and other resources. The OSGi platform allows to install, start, stop, update and uninstall bundles at execution time without the need for restarting the whole platform. Each ***bundle*** expresses its capabilities and requirements in terms of Java packages and other resources. Therefore the platform calculates and manages connections between bundles and assures the satisfaction of mandatory requirements of a bundle before executing it.

The bundles in OSGi, as a set of shared, required and private Java packages or other generic capabilities. OSGi bundles are deployed as a JAR (Java ARchive) file containing a special descriptor file called ***manifest.mf***. This descriptor allows developers to package self-descriptive bundles. The information contained in this description about the bundle's unique identification, version, contents, provided capabilities and the ones it requires from other bundles in order to work and more. Once a bundle is deployed to an OSGi framework, the framework uses this description to resolve declared requirements of the bundle. The resolving process involves matching and linking requirements of the deployed bundle with the capabilities already available on the platform. This process results with the construction of a class space in which the code contained on the deployed bundle will be loaded and executed.

While an OSGi platform manages the lifecycle changes of each module it contains, it does not provide a mechanism for deploying a software system (e.g. an application) with a coordinated fashion. The closest to a deployment system specification is the Deployment Admin Service specification that defines a deployment package as a collection of bundles and other artifacts. The deployment procedure of a deployment package is well defined.

Table 3.3: Comparison of modular platforms

Criteria	CCM	EJB	OSGi
Deployment Unit	Component package, one or more implementations of components, component descriptors, assembly descriptions	Ear, jar archive containing bean implementations and component descriptor files	Bundle, jar archive containing implementations and manifest file
Modularity	Language independent modular execution platform	Java enterprise application execution platform	Java modular execution platform
Site Representation	–	JNDI naming service for accessing resources, other beans	Bundle resources, Service registry
Deployment Activities	Installation, Configuration, Planification, Preparation, Launch, Uninstallation	Installation, Activation, Deactivation, Uninstallation	Install, Activate, Deactivate, Update, Uninstall
Process Hooks	–	–	Listeners on bundle life cycle changes
Distributed Coordination	Coordination based on the concepts of node, assembly and component	–	–
Descriptor File	Component Assembly Descriptor	Different XML files according to component types such as beans.xml, ejb-jar.xml, web.xml	Manifest.mf
Descriptor Placement	Separate descriptors for component description, component package, component assembly	Inside the ear archive	Inside the bundle archive
Policy Description	No custom deployment policies	No custom deployment policies	No custom deployment policies

3.5.5 Distributed Deployment

Computing environments are more and more distributed over multiple machines. Distributed deployment consists of the problem of that conducting the deployment process over multiple distributed machines, connected over the network. Software deployment in a distributed system aggravates the complications and issues discussed in this chapter. One of the main problems is the heterogeneity of resources, which generates the need for modeling different types of target sites, the resources that they contain. The other important issue is the planning and coordination of deployment actions of many components that the software product is composed of. In addition, the issue of dependency management is more complicated because of the physical disparity of target sites and the components that they will host. Last but not least, accessing physical machines dispersed over the network and executing commands remotely is prone to errors and security breaches.

All these issues indicate that it is difficult to deal with the problem of distributed deployment manually. Distributed deployment requires support by some kind of automation tool that should cover as much of deployment activities as possible. The deploy-

ment solutions in distributed environments are studied in three categories as script-based, language-based and model based deployment [Talwar 2005]. In this article, authors argue the trade-offs between these different approaches, represented in this diagram 3.10. As shown, language-based and model-based approaches require more investment to establish but can scale easily and handle deployment of complex systems.

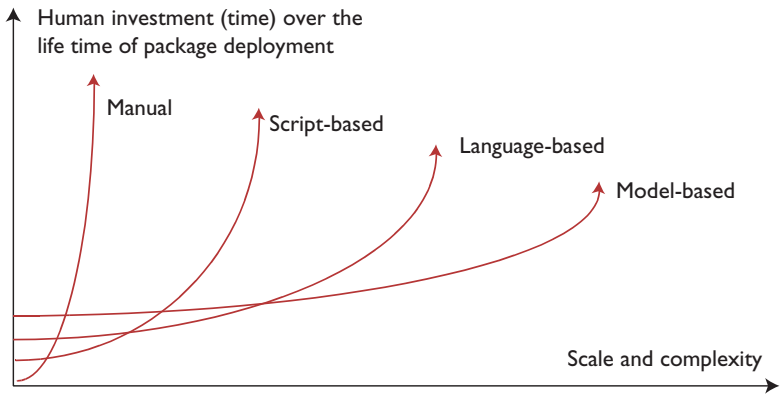


Figure 3.10: Tradeoffs between distributed deployment approaches (from [Talwar 2005])

a. Scripted Deployment

Script-based approach use existing tools and technologies for applying deployment actions on disributed environments. This method makes use of a number of scripts, for example bash scripts) that coordinate existing tools for conducting common deployment activities. These scripts can copy files using tools such as scp over ssh, for applying predefined configuration files. They can also invoke package managers for installing software packages.

At first sight, this approach seems convenient for system administrators who are familiar with these tools. After all it is fully customizable and the process is fairly straightforward. However, it is not suitable for more complex use cases, where managing applications and execution environments with scripts becomes long and difficult to maintain. Site and product models do not exist or are limited to simple ad-hoc models. It has also limited expressiveness regarding to resource description what makes the automation not always achievable. The most important problem with leaving system administrators for deploying systems via scripts is the lack of traceability of their actions. Script-based deployment processes are susceptible of human errors, which can harm the distributed computing infrastructure.

b. Language-based Deployment

Language-based deployment improves on some of the limitations of script-based approaches. This approach uses a configuration language, parsers and tools to perform deployment tasks. A number of deployment methods follow this approach such as SmartFrog [Goldsack 2003] and [Wang 2006]. Specialized deployment language offers an easier

usage for these tools. However, apart from the specialized language the execution of the deployment process is similar with scripted deployment approaches.

Language-based deployment frameworks usually include a distributed deployment management runtime. The language they propose serves to describe the system configuration and the deployment workflow. This language defines an abstraction layer for managing the configurations of deployed software. Using the provided workflow, a dedicated deployment agent can coordinate deployment tasks. A prepared deployment workflow is then executed by the distributed deployment engine that enacts the workflow to achieve and maintain the desired application state.

Using the language-based deployment approach brings several advantages. Mainly, having a language proposes higher-level abstractions for developers that specify the actions of the deployment process, compared to the script-based approaches. This enables associating management strategies like software reconfiguration, automated updates and on-demand deployment. However, language-based deployment modeling does not allow for full deployment automation. The language facilitates specifying the deployment but it is still difficult to associate custom automation policies and enhance the deployment process. With language-based approach it is also difficult to address heterogeneity of resources and components, as the engine that executes the language should still cope with heterogeneous product and site models. These remaining issues are addressed by model-based deployment techniques.

c. Model-based Deployment

Model-based deployment systems leverage architectural models for modeling structure of a software application together with the target execution environment. Architectural models explicitly represent components, connectors, component configurations and their requirements on one side, and execution nodes, network connections and resources on the other. This separation between software and environment models is one of the key advantages of the model-based approach. In such models, the relationship between applications and the target environment are also represented. Usually target environment descriptions include features and resources exposed by the runtime while, the applications, or more specifically composing components, declare their requirements on the former. This improves reusability and enables full automation of the process. The model of a software product can be reused when the software is deployed in different execution environments. Similarly, the model of an execution environment may be reused for deployment of many different applications. Moreover, when component-based systems are considered, the architectural model created during development, can be the basis for a definition of the software deployment model. Therefore, the model-based approach is especially suitable for the component-based systems.

Research community showed special interest on model-based deployment solutions. Software Dock [Hall 1999] Prism [Mikic-Rakic 2002], ADME [Dearle 2004],

DAnCE [Wang 2003], JADE [Bouchenak 2006], DeployWare [Flissi 2008], and DACAR [Dubus 2007] are some of the important examples. These frameworks are based on two key features: A common model for representing the software product and target environment, and a set of model-driven engineering techniques that are used to enhance the common model with different aspects. For instance, Quality of Service (QoS) information can be associated with each component implementation of the software product for a more efficient dependency resolution. With these new aspects, capabilities of the deployment system can be augmented by providing better decisions for deployment activities, be it installation, configuration or update.

Software Dock is a system of loosely coupled, cooperating, distributed components. It supports software producers by providing a Release Dock and a Field Dock. The Release Dock acts as a repository of software system releases. The Field Dock supports a software consumer by providing an interface to the consumer's resources, configuration, and deployed software systems. The Software Dock employs agents that travel from a Release Dock to a Field Dock in order to perform specific software deployment tasks. A wide area event system connects Release Docks to Field Docks.

Prism is a deployment approach based directly on an architectural model. It is destined for resource-constrained, mobile target environments, addressing distribution, heterogeneity and wireless communication issues. Authors present two different modes of deployment process; with and without centralized ownership. In centralized ownership process, a central site continuously analyzes the architectural models of target sites and ensures that they are valid. If it is not the case, the central site prepares a deployment package, with binary components, and sends them to local sites. Each local site is responsible for applying architectural changes and informing the central site once the deployment is successful. In the case of distributed ownership, each local site decides when and what they need in terms of deployment and demands it from the central site.

ADME is a framework for deployment and management of distributed component-based applications. Authors applied an autonomic computing approach using a declarative constraint definition language for specifying high-level goals. Deployment goals are specified in terms of components, deployment sites and available resources. Constraints restrict the deployment process by mapping components to sites and applying topological constraints. A constraint resolver engine evaluates the application configurations and current state of the deployment sites and decides on a mapping between components, deployment sites and connection between those. Deployed applications and deployment sites are monitored by the centralized deployment framework, as so if a constraint is no longer satisfied, the deployment process is relaunched for finding another mapping satisfying constraints.

DAnCE addresses deployment of CCM applications. It is based on the OMG D&C specification that standardizes many aspects of configuration and deployment for component-based systems. DAnCE enhances the D&C data models to describe deployment concerns related to real-time QoS requirements of applications and configurations of middleware services.

DeployWare is based on the Fractal component model and abstracts concepts of the deployment independently of the underlying paradigm and technology. It provides a domain-specific modeling language and a metamodel to mask software heterogeneity. Every notion in DeployWare is being modeled as a component: properties are represented as a composite component that contains the configurable properties of a software, dependencies are composites that contain references to other software components, even procedures, such as install, configure or start, are represented as components symbolizing the instructions. These instructions are runnable components that use the DeployWare libraries to realize elementary deployment tasks.

DACAR is another deployment system that is based on OMG D&C specification for CCM applications. Authors propose using Event-Condition-Action rules for expressing deployment concerns. These rules express what should be monitored on the execution environment (observation rules), how architectural changes are decided (architectural rules) and how the deployments will proceed (deployment rules). These rules are executed in order to construct an autonomic control loop.

Study of these examples show that a multitude of approaches can be associated with models for providing deployment solutions. They use proprietary architectural models or the OMG D&C component model as a common model, but they choose to construct the deployment process using different approaches such as mobile-agents, constraint solvers and rules. Table 3.4 compares three of these frameworks according to evaluation criteria.

3.5.6 Cloud Deployment

Cloud Computing is a model for enabling access to a shared pool of configurable computing resources [Peter Mell and Tim Grance 2011]. It relies on the premise that sharing resources over effectively constructed computing infrastructures would reduce the overall cost of construction, operation and maintenance of software services. Cloud Computing is the result of evolution and adoption of existing technologies and paradigms, such as virtualization, autonomic computing, service-oriented computing and grid computing. Outsourcing computer infrastructures allows companies to benefit from these technologies without the need of costly investments on knowledge and expertise. This helps them to focus on their business, and easily adjust their need for computing resources according to their changing demands.

Table 3.4: Comparison of model-based deployment facilities

Criteria	Software Dock	Prism	ADME
Deployment Unit	Package, containing deployment artifacts and the descriptor	ComponentContent, messages containing mobile code and information about the target location of the component	Bundle, XML-encoded closure of code and data together with bindings naming the data
Modularity	Monolithic software systems	Modular architectural model with components and connectors	Platform that is capable of executing multiple bundles within isolation
Site Representation	Hierarchically organized key-value registry containing information about sites	Partial architectural model of the site	Site configuration in terms of currently running components
Deployment Activities	Release, Installation, Activation, DeActivation, Update, Adapt, DeInstallation, DeRelease	(Request, Receive), Add, Weld, Upgrade, Start	Plan, Install, Instantiate, Wire
Process Hooks	–	–	–
Distributed Coordination	Remote deployment but no coordination	Distributed coordination with centralized or distributed ownership	Autonomic control of distributed hosts based on constraint solving
Descriptor File	Deployable Software Description (DSD), Declarative language	ADL (C2SADEL), transmitted with ArchitecturalModel message	Constraint-based language (Deladas)
Descriptor Placement	Inside the deployment package	ArchitecturalModel messages are transmitted separately from ComponentContent messages	Independent
Policy Description	Different policies can be defined inside the deployment descriptor	–	–

A number of characteristics are identified by widely accepted definition document of NIST [Peter Mell and Tim Grance 2011]. These are ***easy access to standardized mechanisms***, ***resource pooling***, ***multi-tenancy***, ***rapid elasticity*** and the ***measured service***. Looking from the deployment perspective, these characteristics can be resumed into following points that are important for the software deployment process.

- ***Virtualization:*** The main enabling technology for cloud computing is virtualization. The fundamental idea behind the virtualization is to generalize physical infrastructures, transparently mapping those to virtual resources that are easy to use and manage. For software deployment, virtualization helps eliminating the problem of resource heterogeneity by providing uniform interface. In addition, virtual resources can hide some of the complexities of the underlying resources.
- ***Multi-tenancy:*** In Cloud Computing, resources are shared between multiple tenants, and assigned exclusively at run time to one consumer at a time. Assigning

resources is done dynamically based on the consumers' needs. Sharing resources can help increase utilization, and hence significantly reduce the operation cost.

- ***Elasticity:*** Elasticity is the ability to scale in and out by provisioning resources and releasing them. Cloud Computing should provide mechanisms to allow quick and automatic elasticity. The large pool of resources in cloud infrastructures gives the illusion of infinite resources to the consumers, and elasticity provides the flexibility to provision these recourses on-demand.
- ***Volatility:*** In counterpart of elasticity, virtualized resources can be unprovisioned in any time, for leaving physical resources to other demands. To balance the reliability of resources, consumers can provide multiple redundant services for the sake of service continuity.
- ***Monitoring:*** Cloud computing provide mechanisms to measure service usage as well as to monitor the health of services. Measuring services enables optimizing resources and provides transparency for both consumers and providers, allowing them to better utilize the service. Measured services can help in building closed-loop cloud systems that are fully automated.

Cloud computing promises agility to the consumers, by giving the ability of provisioning on-demand services. Cloud computing providers offer their services according to three major service models.

Software-as-a-Service (SaaS) refers to the service model in which a service is a software service that allows the consumer (end user) to access and use a provider software application that is hosted, deployed, and managed by the provider. Consumers have limited control over the application, and are restricted in how they can use and interact with the application. The application is usually accessed via a thin client (i.e., Web browser), through which consumers can input data and get output. Examples of SaaS are email services (i.e., Gmail), business applications such as customer relationship management applications (i.e., Salesforce), and data storage services (Hosted SQL or NoSQL Databases). Because consumers have limited control over SaaS applications, this service model has little interest for the software deployment process. Nevertheless, any deployed software can depend on some SaaS, so the deployment coordination may involve sending appropriate configuration to a SaaS.

Platform-as-a-Service (PaaS) refers to the service model that offers a platform service on which consumers can define, develop, configure, deploy, manage, and monitor cloud software. Mostly, PaaS provides a managed infrastructure and low-level software (operating system and an execution platform) on which consumers can build their software. Although consumers can control their deployed software, PaaS providers do not give direct control over the underlying cloud infrastructure.

Instead consumers are given the choice of customizing the platform service with access to physical resources and other software services such as event distribution and data storage. Windows Azure, Google App Engine and Heroku are examples of PaaS providers.

The advantage of such platforms for deployment is that they abstract the communications with the lower-level infrastructure and provide easy to access and easy to use interfaces for managing software deployment. Independent from the technology of the underlying execution platform, deployment activities can be commanded via graphic user interfaces (GUI) or automated via application programming interfaces (APIs). In addition, some PaaS providers allow consumers to push code directly to the platform using distributed source revision systems such as Git.

Infrastructure-as-a-Service (IaaS) refers to the service model, which allows the service consumer to lease infrastructure capabilities based on demand. The infrastructure capabilities include processing, storage, network, or any other basic computing resources that can be used to deploy and run execution platforms (i.e., operating systems, management tools, development tools, and monitoring tools) and the applications developed on top of the platforms. IaaS consumers are not given direct access to resources but have the ability to select and configure resources as required based on their needs.

IaaS is very close to the virtualization, since it serves from virtualization technologies to partition physical resources, in order to provide the consumers with a pool of storage and computing resources. Indeed in many cases consumers are provided with a preconfigured operating system. Operating system configurations are created as hard drive images of a system snapshot. Using virtualization techniques enables provisioning rapidly different ***instances*** of this machine image. This paradigm is interesting for the software deployment, because it lets developers release software bundled into images, preconfigured with the execution environment, ready to execute. For instance, Amazon Elastic Compute Cloud service accepts ***Amazon Machine Image (AMI)*** as deployment unit to provision virtual machines.

Lastly, the physical infrastructures of Cloud Computing providers can be installed in different environments. There are different types of deployment environments are known as deployment models and differ according to the physical location, the platform constraints and overall access to the facilities. A ***public cloud*** refers to an installment of physical infrastructure facilities that are provided by a third party. The public cloud is shared between multiple organizations or consumers. It is the least expensive amongst other models but suffers from the lack of a trust model between providers and consumers. On the contrary a ***private cloud*** is owned by a cloud provider but installed entirely on the premises of consumer, which is the software service provider. This eliminates the trust issue and provides more flexibility, as organizations can implement their own privacy, security and access policies. However, this option is the most expensive one in terms of cost

of operation and maintenance. There are ***hybrid cloud*** models that makes compromises between public and private models by establishing trust policies between providers and consumers, partitioning critical resources and other support services.

3.6 Conclusion

This chapter presented the notion of software deployment; the process of delivering the software from its production, i.e. the development, until its execution. Contrary to the early assumptions, the deployment is a process, which involves a set of correlated activities for configuring and bringing the software to its desired state at execution. This process can continue along the lifetime of a software system, forming a bridge between the development to the execution. The bridge between two worlds gains more and more importance in recent years, with the increasing need to manage software evolution at execution via software reconfigurations and adaptations.

This chapter presented some of the important issues that need to address in order to provide deployment solutions. To tackle these issues software deployment domain is in close relationship with other domains of software engineering. From software architectures to software product lines and self-adaptive systems, these domains not only showed solutions for these issues but also expanded the scope of software deployment.

Involving many actors, the deployment process includes not only tools for automating activities but also a set of models and process definitions that serve to model and optimize deployment tasks. Corresponding to this view, this chapter introduced the notion of software deployment facilities as the group of models, processes and tools employed by an organization for handling deployment processes by optimizing and automating its tasks. Later, many examples of deployment facilities from different approaches have been introduced briefly and compared against a set of evaluation criteria.

To conclude this chapter, there are two major reasons for that automated software deployment facilities are needed more than ever. First, automated processes are needed for deployment because software development keeps accelerating and software producers need to push changes into execution environments as rapidly as possible. Second, propelled with the emergence of new computing domains such as cloud computing and pervasive computing, dynamically evolving systems makes it impossible for producers to deploy software by hand, with human processes. Instead software deployment processes automated with appropriate tools should continuously deploy software into execution, reacting to the dynamic changes in environments and requirements. This newly applied paradigm, continuous deployment is the subject of the next chapter.

Chapter 4

Continuous Deployment

"Loneliness doesn't come from having no one around you, but from being unable to communicate the things that are important to you."

— Carl Jung

Contents

4.1 Introduction

The goal of software deployment is to construct executing software using artifacts created in the development. The previous chapter introduces the software deployment process in detail and discusses the need for its automation. It concludes by stating that automated deployment facilities are needed to cope with accelerating software development and dynamism of novel execution environments. This chapter focuses on a recent trend, radically changing the way software is deployed: continuous deployment.

Before going into details, let's first define continuous delivery and continuous deployment. ***Continuous delivery*** (*Cd*) is a set of practices that transforms the software development lifecycle. It can roughly be summarized by the phrase "*Every commit triggers a release.*" So every change made by a developer is integrated into a new software release, ready to be installed. ***Continuous Deployment*** (*CD*) extends this principle to the actual deployment of the created release. Pushed to its limit, it means that every commit is pushed to production.

Obviously both *Cd* and *CD* require rigorous methods and sophisticated tools. There are many different ways to achieve *Cd* and *CD* but most of them rely on the idea of ***deployment pipeline***. This pipeline represents the journey from the development to the release repository or the production environment.

The goal of this chapter is twofold. First, it aims to introduce the general idea behind continuous deployment. For that matter, this chapter starts by presenting the concepts of lean development and deployment pipeline. Then in the following section it discusses the current practices that are used for implementing deployment pipelines. The concerns addressed by these technologies involve a large range of software lifecycle phases that exceed the subject of this thesis. This is why the second part of this chapter focuses particularly on the execution platforms and the deployment process. It studies the requirements for implementing deployment facilities that support continuous deployment. Each one of these requirements is detailed in order to establish a characterization framework for continuous deployment facilities.

4.1.1 From Lean Development to Continuous Delivery

The Agile Manifesto affirms that responding to change is more important than following a strict project plan. Development processes evolving around this vision acknowledge that changes are inevitable throughout the project and that investing in immutable system designs is counterproductive. However, this does not mean that the software producers must compromise on the quality of the software systems and the rigor of the process that produces these systems. On the contrary, different stages of the software lifecycle require optimizations more than ever, to be able to cope with the change and still provide quality software.

Lean software development (LSD) is the application of Lean manufacturing principles to software lifecycle processes. Lean as a manufacturing and production practice, aims to *create value with less work*. The ***value*** concept is defined as any action or process that brings added value to the product or service. Lean manufacturing is based on optimizing value-creating flows in order to increase efficiency and decrease the waste. The goal of LSD is to reduce the time and effort wasted for producing, releasing and deploying software. This is enabled by setting rigorous practices and processes that continuously reflects produced value over the software product. Continuous Delivery applies this principle by turning every value created by developers to a software release, all by guaranteeing the quality of the released product.

Kanban, presented briefly in the previous chapter in the section 3.1.2, is a method for keeping track of the work-in-progress and managing flow. It provides a good framework for organizations to apply lean principles [Poppendieck 2012]. The Kanban board represents the stream of values that are being created. The ***value stream*** represents the work-in-progress values that pass through different states and enter into the responsibility of different teams (see figure 4.1). The key to a Kanban system is that within any value-adding activity the total amount of active work is limited. Therefore the entire value stream contains a limited amount of work. This pushes the incentive for the teams to consider optimizing and adapting the value stream activities to avoid bottlenecks.

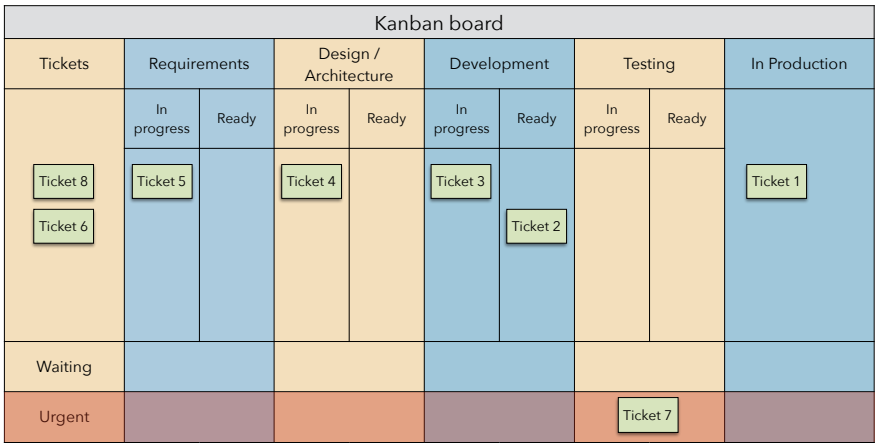


Figure 4.1: Kanban Board

4.1.2 Value Stream in Software Lifecycle

The ***value*** produced in software lifecycle manifests itself under different forms. The most obvious type of software value is the ***source code***, which is built into executable binaries. But most of the time binaries are not enough for executing the system. Correct execution of a software system requires other assets. One type of value that is usually needed is the set of ***configurations*** to execute the software. Software systems execute on software and hardware infrastructures that are meticulously determined in order to guarantee the

correct execution of the system. Operations teams create these execution ***environments*** as descriptions or the disk images. Finally, applications usually require ***data***. For example, the database schemas are values produced during the development. The figure 4.2 depicts these four types of value.

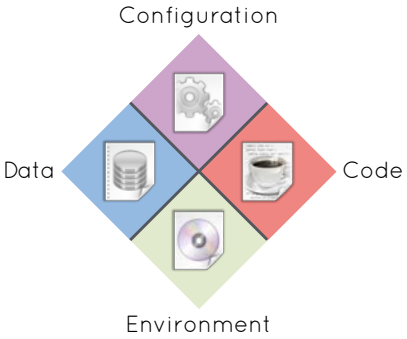


Figure 4.2: Software Values

To be able to track the evolution of the system each value, changes made to these assets must be versioned and archived. A ***commit*** represents the identified, versioned record of change that is brought to the system. In a continuous delivery (*Cd*) system every commit – therefore added value – triggers a set of processes to test and validate the change and produce a release of the system. Continuous deployment (*CD*) pushes this further by deploying each commit to the production environments.

The ultimate goal of software producers is to deliver high-quality, valuable software in an efficient, fast and reliable way. Rationally each commit passes through different phases until it is a part of the running system. The term ***cycle time*** refers to the time it takes inside an organization from deciding to make a change in the system to making it available to the users. When a new feature is developed or a bug is fixed by the development team, it passes through a set of ***quality gates***, which validates its conformity and quality. The system is ready to be delivered to the consumers, once an overall quality and confidence is guaranteed.

A lean delivery process encourages development teams to work in an empirical approach, in which they can test new ideas and get early feedback from automated tests and customers. *Cd* and *CD* are enabled by a streamlined process that continuously evaluates the reliability of the software system at each commit, such that the latest reliable version of the software system is always available. The following section presents the ***deployment pipeline*** that implements this process.

4.1.3 Deployment Pipeline

The deployment pipeline proposes a solution for the problem of continuous delivery. It provides an end-to-end approach to delivering software by automating all the processes from version control until execution. In this pipeline every change to the software goes

through a complex process on its way to being released. The process involves building the software, followed by the progress of these builds through multiple stages of testing, deployment into different environments and finally the release [Humble 2010].

Definition 5: Deployment Pipeline

A deployment pipeline is a holistic process that automates certain software lifecycle activities such as build, test, deployment and release; which enables tracking each value from the it's conception at development until it's inception in the system.

The deployment pipeline provides visibility into the production readiness of software by observing and controling the progress of each change through different activities. As presented in the previous chapter (see 3.2.5), software lifecycle requires the involvement and collaboration of many different actors, such as developers, testers and operations personel. Having a holistic deployment pipeline enhances how many individuals from different teams work together effectively.

The deployment pipeline is a pull-based system [Poppendieck 2009]. Rather that pushing changes to different actors, the changes produced by the developers are built and stored in artifact repositories. This way changes are built once and generated artifacts can be associated with the change version. Only then different actors such as testing teams and operations can pull these builds as they need in the continuous flow. The central enabler of the pipeline is a repeatable, reliable and automated deployment process that produces deterministic results.

Using this deployment process, overall cost and risk of releasing and deploying software is reduced. Quality assurance teams can pull and deploy builds into testing environments. Similarly, operations can deploy builds into staging and production environments. The figure 4.3 depicts a generic deployment pipeline. The actual implementations of this model depend on the structure and requirements of the organization.

Since the deployment process (whether to a development machine or for production) is automated, it can be executed and tested regularly. Indeed for each change (on code or configuration) there can be a deployment on a testing environment. As a result, involved teams can get rapid feedback on the code and the deployment process. The idea of transferring knowledge regularly from the deployment process to the development team gave rise to a new movement called DevOps [Humble 2011]. The idea behind DevOps movement is to encourage the close relationship between different actors; developers, operations and testers (or quality assurance), involved in the software production, who belong traditionally to different backgrounds. The continuous feedback provided by deployment pipeline bring together the developers, who are in charge of requirement analysis, design and development with operations teams, who supervise the deployment, execution and maintenance.

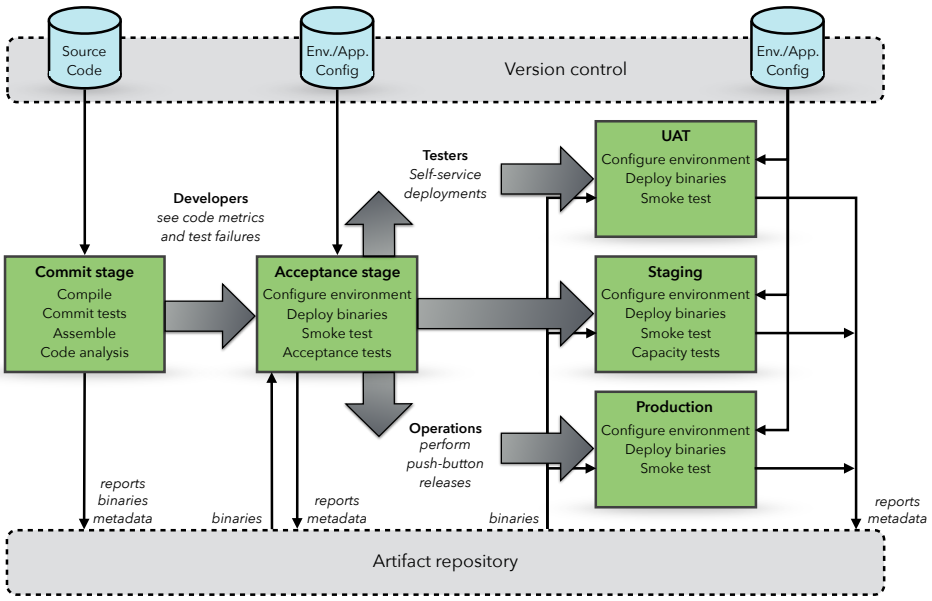


Figure 4.3: Deployment Pipeline (adapted from [Humble 2010])

4.2 Enabling Technologies for Continuous Deployment

The continuous deployment, as seen in the previous section is built upon a set of best practices around the pivotal concept of deployment pipeline. The deployment pipeline is made possible by a set of tools and indeed, accommodating right tools helps to establish best practices inside an organization. This section presents briefly the practices that are essential to establish a continuous deployment pipeline. These are source code management, automated build, continuous integration, artifact management, automated deployment and autonomic control loop.

4.2.1 Source Code Management

Version control systems (VCS), also known as source code management (SCM) systems, are a mechanism for keeping multiple versions of files, so that each modification is archived and previous versions of files are accessible. Beyond that, they are tools through which people involved in software delivery can collaborate. These tools provide a workspace for collaborating and creating new value from existing ones.

VCSs help teams to avoid (and resolve) conflicts that may appear during development by keeping track of changes made on controlled files. Indeed, this is valid not only for source code files but for every single artifact related to value creation. These artifacts may include source code, documentation, configuration files, files related to the build or even virtual machine images. Each change to these files is identified by a commit that represents a a particular version of the software. More recently, with the emergence of

systems that automate deployment process, the deployment configuration of applications and the description of the runtime environments are equally included in version control as code. The trend to treat those artifacts as code and put them into version control is known as the *infrastructure-as-code* [Spinellis 2012].

There can be different types of VCSs depending on the type of the stored artifacts. Git[1] is an example VCS for source code and configuration files. Distributed VCSs like Git provide entire history of the source repository to each user. They allow developers to easily create local and remote branches, work offline, merge changes and push them to other users. They also enable advanced collaboration scenarios such as code reviews.

4.2.2 Automated Build

A build process is a sequence of tasks that transforms development artifacts (source code, configurations etc.) to deliverables (executable binaries, distributions etc.). Depending on the technology, this process can include steps such as dependency calculation, environment-specific compilation, different kinds of tests and packaging.

While the value is created as raw artifacts, the focus of a deployment pipeline is deliverables. This is why it is crucial for the coherence of the pipeline that the build process is automated in a way to produce deterministic deliverables. Automated tests attached to the build process verify each build in order to detect errors as soon as possible. Tests are particularly important, because they validate the correctness and quality of resulting deliverables. They are focused on asserting that the code compiles successfully and passes a body of unit and acceptance tests. There are many tools that allow build automation, such as Make[2], Apache Ant[3], Apache Maven[4], Gradle[5], and MsBuild[6].

4.2.3 Continuous Integration

Continuous Integration is a software development practice where members of a team integrate their work frequently, leading to daily integrations of projects. Frequent integrations are enabled by an automated build process that rebuilds and tests the system at each change. Continuous integration ensures that teams working together to create complex systems can do so with a higher level of confidence.

A continuous integration tool provides a bridge between different other tools. It is based on the notion of ***job*** that executes and integrates different tools (see figure 4.4). A change in the source code, a dependency or a fixed period can trigger the execution

[1] Git: http://git-scm.com/
[2] Make: http://www.gnu.org/software/make/
[3] Apache Ant: http://ant.apache.org/
[4] Apache Maven: http://maven.apache.org/
[5] Gradle: http://www.gradle.org/
[6] MsBuild: http://msdn.microsoft.com/en-us/library/dd393574.aspx

of jobs. Usually the job execution merges the changes from the VCS, invokes the build process and test suites. Indeed in a deployment pipeline, it is crucial that every change (every commit to the VCS) triggers a build process in order to detect early conflicts and errors. This allows teams to follow the state of the software product at each commit. If the build process and all the tests finish without errors, the job ends by producing binaries and often by publishing them into an artifact repository. It is essential in the context of a deployment pipeline to produce executable binaries exactly once for each commit and keep the track of which commit produced which set of binaries.

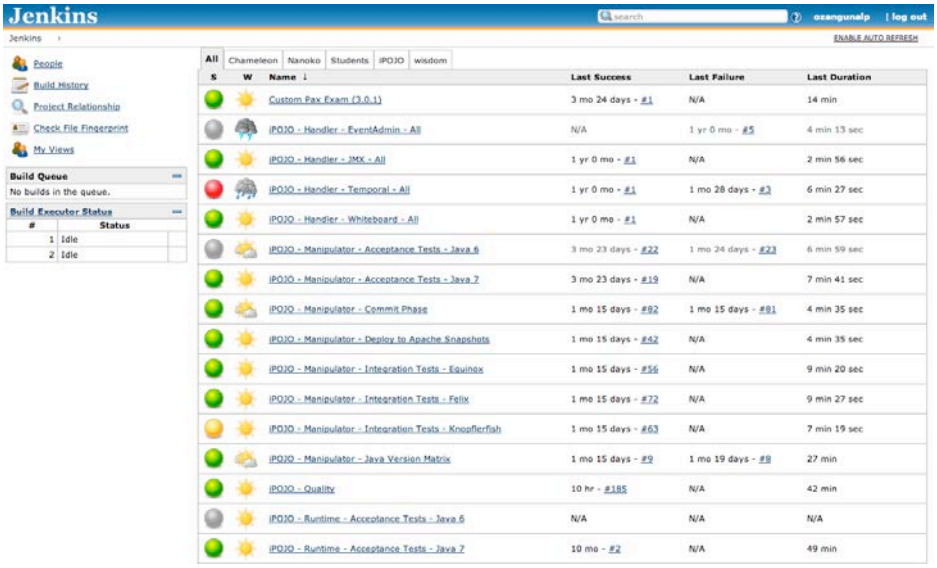


Figure 4.4: Jenkins Job List

Another type of job can connect code metrics and profiling utilities that evaluate source code and generate reports about its quality. However, for the rapidity of builds, the build and test process should not be exceedingly long. Usually 10 minutes is an acceptable time for project build and unit tests to be completed.

Continuous integration mainly focuses on development teams. The output of the continuous integration system, therefore the binaries and build reports, normally forms the input to the manual testing process and to the rest of the release process. The goal of the deployment pipeline is to continue automating the rest of this process. There are many tools that lets implementing continuous integration such as Jenkins CI[7] and Travis CI[8].

[7]Jenkins CI: http://jenkins-ci.org/
[8]Travis CI: https://travis-ci.org/

4.2.4 Artifact Management

Artifacts are assembled pieces of software that include packaged, deliverable application code, application assets, virtual machine images, and (typically) configuration data. An artifact is identifiable with a unique name and version. Each identified artifact is also immutable. An artifact repository manager stores and organizes artifacts and metadata about those. Repository managers are capable of archiving multiple versions of artifacts and analyzing those according to policies indicating product quality such as dependability or performance. They allow to publicly share binaries with members of the team or third-party collaborators. Sonatype Nexus[9] is an example of repository managers that is widely used by developers to archive and publish software artifacts. Another example of repository manager is a service called Bintray[10]. It provides social services for developers to store, publish, share and download software artifacts, and receive feedback over users of their packages.

4.2.5 Automated Deployment

As expressed above, deployment automation is a must for the deployment pipeline. The previous chapter studied software deployment in details, and presented different approaches to automate the deployment process. Throughout the course of the deployment pipeline the deployment process is invoked in different stages for deploying applications into testing, staging and production environments.

The deployment automation takes care of two distinct but complementary deployment requirements. Firstly, teams need to create runtime environments on-demand by using tools that can provision machines (possibly virtual machines) and configure those with prescribed environment templates. Such environments are usually software stacks consisting of the operating system, required services and the middleware. Vagrant[11] and Docker[12] are examples of tools that allows defining runtime infrastructures. Then, secondly, using the provisioned environment as a basis, applications can be deployed by retrieving binaries from the artifact repository and application configurations. Such tools include Chef[13], CFEngine[14] and Puppet[15].

The whole process of provisioning the runtime environment and deploying the application must be automated, easy to invoke and ***deterministic***. For example, if a script is realizing the application deployment, it must be accessible to everyone through version control and give the same result in same conditions no matter where it is deployed. This

[9]Sonatype Nexus: http://www.sonatype.org/nexus/
[10]Bintray: http://bintray.com/
[11]Vagrant: http://www.vagrantup.com/
[12]Docker: http://www.docker.com/
[13]Chef: http://www.getchef.com/
[14]CFEnginge: http://cfengine.com/
[15]Puppet: http://puppetlabs.com/

is required to make sure that the software tested throughout the pipeline is the same as the system released into production. Together with the automated build, this property ensures that artifacts built on the development machines are the same as the one that reaches the production.

The deployment pipeline depends on the ability of different teams to reproduce runtime environments and applications. Each time a commit passes the build process and automated tests, a well-defined deployment process can be invoked automatically. The automated deployment process pulls the latest changes, deploys and executes the software for acceptance tests and staging. Here the software system runs on configurations as close as possible to the production. It is subjected to different kinds of functional and performance tests. Tools that test application behavior include Apache JMeter[16] and Cucumber[17] , whereas LoadUI[18] is a tool for performance testing. Only if the functions are qualified, the new system containing the changes of the commit can be released to the production.

It is possible to implement different release policies for the delivery of the software into the production environments. *Cd* and *CD* are distinguished in this respect. If the deployment pipeline applies *CD*, the deployment into production occurs at each commit. A common practice is to deploy changes to production by ***promoting*** staging environments into production. This way deploying a new release, for example of a web application, can be as easy as switching requests from old production machine to the new one. Other deployment patterns include canary releases, where multiple versions of the same product coexists or yet, blue/green deployments in which new release takes over to the old one gradually.

In these use cases automated deployment must not only conduct deployment actions in one deployment host, but also coordinate deployment processes in multiple hosts. Remote administration tools such as Capistrano[19] and Fabric[20] help realizing such tasks. However, in some environments such redundancy is not permitted. For example, in pervasive environments machines are not replaceable due to physical location concerns. Instead, new releases should reconfigure and adapt existing production environments in order to deploy the new release.

4.2.6 Monitoring & Control Loop

In lean software development it is pivotal to continuously improve applied processes by rapid feedbacks. Throughout the deployment pipeline feedbacks are constantly gathered from build and test reports. When a problem is detected, the instance of the deployment

[16] Apache JMeter: http://jmeter.apache.org/
[17] Cucumber: http://cukes.info/
[18] LoadUI: http://www.loadui.org/
[19] Capistrano: http://capistranorb.com/
[20] Fabric: http://www.fabfile.org/

pipeline is stopped and error reports are transferred to the developers. On the other hand, in addition to different kinds of tests (unit, integration, acceptance, functional, etc.) it is also important to monitor running systems for errors and unexpected behavior. As soon as a problem is detected on the production system, development and operations teams act on to resolve the issue and push a change to the deployment pipeline that fixes it.

Analyzing system logs is still one of the essential ways of spotting problems at runtime. For systems running on multiple machines, logs should be collected from each machine, stored and indexed for search. For system monitoring, tools such as Nagios[21] and Collectd[22] provide metrics and auditing on system health and performance. For monitoring applications, each middleware technology provides touch points for monitoring the aspects they handle. More generic approaches also exist. CIM [Dis 2014] is an open standard that provides a common definition of management information for systems, networks, applications and services. It enables consistent information exchange between multiple parties about managed systems. JMX [Sun Microsystems 2006b] specification provides a standard way for monitoring and managing Java Virtual Machine. The .Net Framework provides monitoring management interfaces through Windows Management Instrumentation (WMI) [Microsoft 1998].

In spite of many possibilities for monitoring running systems, it is still difficult for operations teams to make decisions and take actions depending on low-level information such as logs. Graphite[23] is a visualization tool that aggregates different monitoring metrics and provides meaningful graphs to system administrators. In most cases operations teams – including system administrators – need to run in-depth system diagnostics to detect root cause of errors. Such diagnostics may require a *post-mortem* analysis of the system state (memory, disk space etc.) at the time the error has occurred. Similarly, system optimization requires comprehensive reports over a period of time, in order to detect performance bottlenecks, security soft spots and memory leaks. In order to obtain high-level indicators for the system's health and performance, system administrators need to store monitoring data historically and then run analysis on it.

Once operations teams are capable of analyzing running systems and detecting errors, they can take relevant actions in order to circumvent errors and/or to optimize the system. Most of the time these actions involve a deployment process of reconfigurations and updates. Similar to the monitoring, these actions are effectuated through a number of touch-points from different layers of the system. The key for achieving *CD* is to minimize the time between the problem detection and the propagation of the solution that fixes the problem. Therefore for fast response times system operators need to establish a control system by automating both the monitoring and the deployment actions.

The idea of ***autonomic computing***, initiated by IBM in a manifesto [Horn 2001], pro-

[21] Nagios: http://www.nagios.org/
[22] Collectd: https://collectd.org/
[23] Graphite: http://graphite.wikidot.com

poses a solution for this problem. Autonomic Computing aims to develop self-managing software systems in order to minimize human intervention during their operation. IBM identify four properties for self-managed, autonomic systems:

- ***Self-configuration:*** The system has the capacity to configure itself and its components in an automated way, guided by high-level policies and goals.
- ***Self-optimization:*** The system and its components have the capacity to optimally use the available resources. It continually seeks to improve its own performance and efficiency.
- ***Self-protection:*** The system automatically defends against malicious attacks or cascading failures. To protect itself, the system must detect (or anticipate) risky situations by using early warning and prevent systemwide failures.
- ***Self-healing:*** The system automatically detects, diagnoses and repairs software and hardware problems. The goal of the system is to increase fault-tolerance and the availability of the system and its services.

Autonomic systems are made of collections if autonomic elements, which manage its own behavior and its relationships with other elements, in accordance with policies established by humans and other authorities. Inside an autonomic system, elements that are managed can be in many levels. At low-level, the managed element could be a hardware resource, such as storage, a CPU, or a printer. At this level autonomic abilities of individual components are relatively limited and hard-coded. Particularly well-established techniques are used for providing fault-tolerance. At higher levels, software resources of different scales can also be managed, such as databases, legacy systems, software components, application services and third-party software utilities. Usually software management allows increased dynamism and flexibility. Autonomic aspects can be expressed in more high-level, goal-oriented terms, leaving the rest to the autonomic capabilities of the element.

Autonomic elements need to continuously sense and respond to the environment in which they are situated. Constructing autonomic elements requires to implement, implicitly or explicitly, one or more feedback loops that will gather information and act on the system and its environment in order to meet the established goals. In a classical sense a control loop includes at least three steps: information gathering, decision=making and action. Based on the information collected in the environment and its internal state, the system determines the necessary actions to comply with a set of objectives and acts accordingly.

IBM proposes a logical architecture for the implementation of this control loop, identifying four distinct activities performed by an autonomic loop control [Kephart 2003]. The reference model for an autonomic manager is composed of five parts (see figure 4.5), known as ***MAPE-K***:

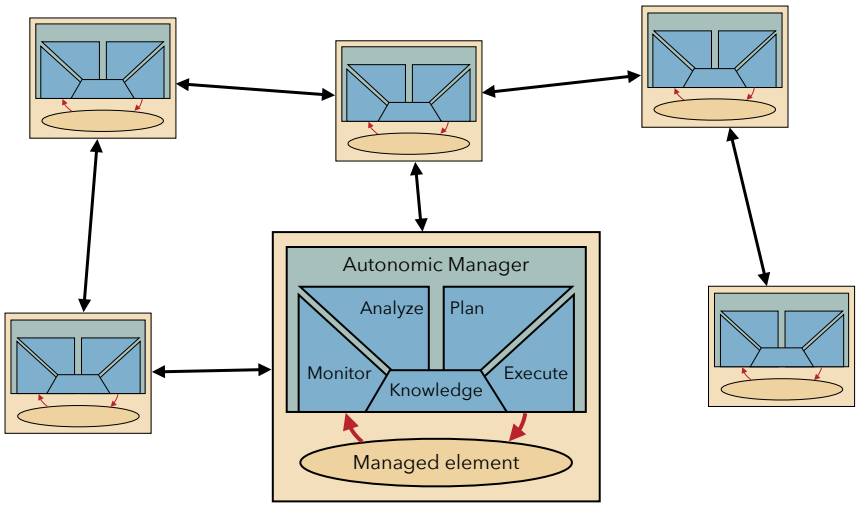


Figure 4.5: Autonomic Managers

- ***Monitor:*** This module is in charge of collecting information coming from monitoring touchpoints. It prepares an activity report by filtering and aggregating the gathered information. Usually it takes into account information in temporal windows for including temporal relevance of gathered data.
- ***Analyze:*** This module determines if changes are required according to the reports provided by the monitoring. It can diagnose problems by detecting correlations and anticipating situations that can occur. As a result, it produces a list of problems to resolve.
- ***Plan:*** Planning resolves the identified problems. It accomplishes this by proposing an action plan, a series of operations that allows to attain a particular state, that will resolve the problem.
- ***Execute:*** It applies the problem resolution. It acts on the managed element using action touchpoints for applying actions fixed by the plan.
- ***Knowledge:*** Knowledge base allows different modules to exchange messages and persist the information they produce, for example, measured values or a history of these last, as well as information on policies and high-level objectives. It is continuously updated to trace the evolution of the state of the system.

The foremost advantage of this separation lies in the increased maintainability and reusability. Modular design and well-defined interfaces between different modules, as recommended by the reference model, allows to use different tools and technologies for each activity. These tools can be implemented separately by different companies or groups who can focus on a particular aspect.

4.3 Requirements for Continuous Deployment

Previous sections of this chapter introduced lean software development and discussed the practice of continuous deployment in more details. The remainder of this chapter concentrates on the continuous deployment in modern, dynamic execution environments, such as pervasive and Cloud computing. This section discusses the requirements for constructing deployment facilities that are able to apply continuous deployment in dynamic execution environments. The previous chapter of software deployment evaluated certain existing works against a number of criteria (see section 3.5.2), which are derived from the characterization framework of [Heimbigner 1998]. The requirements studied in this section aim specifically to identify challenges brought by dynamic environments to the problem of continuous software deployment. The most apparent characteristics of such environments are the unanticipated, continuous change and the fact that deployment requests can originate from different sources. Similar to the previous chapter, here also the requirements are studied in three categories, requirements linked to the deployment platform, requirements regarding the deployment process, and the requirements for the language that describes the deployment. A preliminary version of this study is published in [Escoffier 2013b].

4.3.1 Platform Requirements

The first category of requirements focuses on the deployment platform, i.e. the capabilities that must be provided by the execution environment to support the continuous deployment of pervasive applications. First of all, the platform must be able to deploy components separately at runtime. Because of the dynamism exhibited by pervasive environments, the platform must also provide configurability, reflection, architectural reconfiguration and context access capabilities.

a. Configurability

The deployment process is not limited to transfer software resources to the execution environment; it also includes configurations. This requirement is particularly important in pervasive applications as the configuration is one of the most used levers to handle adaptations [McKinley 2004]. Because of the dynamic adaptations required by pervasive systems, the configurations must be updatable at runtime.

- ***Execution Platform Configurations:*** The configurations to initialize the execution platform at start-up.
- ***Load-time Configuration & Composition:*** The ability to configure code at load-time. For modular platforms this may also include the composition and binding configurations of modules and components.

b. Modular Dynamic Execution Platform

In order to adapt themselves to unanticipated situations, pervasive applications must be modular [Kramer 2007]. The platforms and applications must be composed of distinct modules that can be installed, updated and uninstalled individually. Modularity may also involve dependency management. Modules could declare dependencies that must be resolved by the deployment process. The targeted entities can be another module, or be more abstract in order to introduce variability and constraints.

- ***Loading/Unloading code:*** The ability to both load and unload modules dynamically at runtime.
- ***Dependency expression and resolution:*** The type of dependency expressions and how they are resolved and satisfied on the execution platform.
- ***Dynamic programming languages:*** Some execution platforms allow dynamically loading code using dynamic programming languages.

c. Reflection

Managing modules and configurations is not enough. The platform must also provide information about the current modules, their states, and configurations, and allow to change these when needed. The capability to change represented aspects, i.e. intercession, is a must for the deployment process to modify the execution platform. Having introspection facilities is an also absolute requirement to let the deployment process determines changes to apply on the system. The introspection data must not be limited to deployment information, but also represent system specificities, available services, computational resources and any data required by the deployment agent to drive the deployment process successfully.

- ***Structural reflection:*** The platforms capability to reflect its static structure, such as modules, packages, configurations.
- ***Behavioral reflection:*** The platforms capability to reflect its execution such as processes, variables and threads.
- ***Call interception:*** The platforms ability to intercept calls, such as method invocations and variable accesses. The interception can change the content of the invocation.
- ***Binding interception:*** The platforms ability to change and customize the default policies for binding module and component dependencies.

d. Architectural Reconfiguration Support

The dynamism of the pervasive environment influences the architecture of pervasive applications [Oreizy 1999]. For instance, a new resource can become available, while another disappears. These events can involve dynamic architectural reconfiguration. In order to make these adaptations possible the platform must provide primitives on which the deployment agent can rely.

- ***Architectural script:*** The execution platform can provide basic operations and scripts on top of those in order to enable architectural reconfiguration.
- ***Service-orientation in component binding:*** The service-orientation in component interfaces and binding mechanisms allows architectural reconfiguration to be seamless during execution.
- ***Tranquility:*** The platform can provide tranquility (sometimes called quiescence) of components during architectural reconfiguration. This may involve deactivating the component, saving its current execution state and restoring that state at the end of the reconfiguration.

e. Context access

The fluctuations of the environment in which the pervasive applications are executed are also posing problems in terms of deployment. Contextual changes have an effect on the deployment process. To implement the continuous loop required to deploy pervasive systems and keep them effective when facing new situations, the deployment agent must be able to monitor the environment. For this reason, the platform must offer context mining, and observation features. This requirement does not define the scheme or the type of represented context data. However, every piece of information required to optimally drive the deployment process should be accessible by the agent.

- ***Flexible context model:*** A flexible context model can be extensible with new context information and clients of this model can choose to consume different views.
- ***Context mining:*** The context mining involves the ability to discover new information from the represented context. It may involve correlation analysis of certain context data inside a temporal window.

4.3.2 Process Requirements

The second category of requirements focuses on the deployment process itself. Dynamic environments impose several characteristics to the deployment process. Whether it is for an installation, update or uninstallation, the deployment process is initiated either from

the system or externally. Then, it analyzes the deployment request and defines a deployment plan listing all the actions. This process includes the selection and/or refinement of the components to deploy. Because of the pervasive environment characteristics, the decisions taken during the deployment process may become invalid, and adaptations must be applied to keep the applications in an operational state.

a. Pull/Push

The deployment process may be triggered either by the system itself or push from externally. In the first case, the system discovers a new required resource, such as a device driver and asks the deployment agent to install the required artifacts. In the second case, the deployment process is triggered by an external entity. It can be the user having purchased a new application on a store, an application update pushed by the application vendor or the platform operator updating technical services. The openness and uncertainty of the pervasive environment requires that both the pull and push modes be supported. More interestingly, the source of the push is not unique. Having multiple sources makes the scheduling and prioritization of deployment requests more complex.

- ***Push:*** The deployment request is pushed to the platform start a deployment process. Multiple deployment processes on remote machines can be coordinated with the push approach.
- ***Pull:*** An internal change from the platform can trigger a deployment request that pulls the changes to deploy and executes the deployment process locally.
- ***Both:*** The deployment request can be pushed from an external authority or pulled by the initiative of the platform.

b. Determinism & Idempotence

Determinism is an essential property to make pervasive system deployment reproducible. For a particular environment, on a specific platform, a singular deployment process must always result to the same system. Such a capability is critical for making the deployment process testable, and improves the reliability of the deployment infrastructure.

Idempotence implies that the deployment will not change the system if it is already in a desired state. This is important because running the deployment process multiple times will only change resources that are necessary to change, without touching others. This property is rarely supported in traditional deployment tools. Unfortunately, the multiplicity of deployment sources makes the idempotence a requirement necessary but difficult to satisfy.

- ***Idempotence:*** The actions taken during the deployment and the overall deployment process is idempotent. For example, deploying an already deployed artifact won't modify the system.

- ***Deterministic:*** The decisions made during the deployment process are result of deterministic processes and algorithms.
- ***Probabilistic:*** The deployment facilities employ probabilistic algorithms during deployment.
- ***Heuristic:*** The process use heuristics to decide on the actions taken during the deployment.

c. Fault-tolerance

The deployment process is constituted from a set of actions that change the pervasive system. However, one or more of these deployment actions can fail. In this case, it is essential to rollback to an operational state, avoiding stale situations. As a consequence, all the deployment activities must be executed inside a transaction [Coghlan 2005]. Many deployment technologies are supporting transactions, however in case of dynamic environments transactions are not only impacted by the deployment process but also by external events. This aspect makes the transaction support very complex to implement.

- ***Concurrency Atomicity:*** Two concurrent deployment processes does not effect one another. This is also called isolation.
- ***Failure Atomicity:*** A failure atomic deployment process performs either a deployment entirely or, in case of a failure, not at all.
- ***Consistency:*** A consistent process assures the integrity of the constraints on the platform state. These are invariant properties of the platform resources and deployed applications.
- ***Durability:*** Durability property states that if a deployment process succeeds, the changes it brought to the platform are permanent, until another deployment process. In certain environments, where some changes are contingent and uncontrollable; assuring durability is not an option.

d. Customizability

One of the main differences between traditional deployment and pervasive system deployment is the unknown environment in which the applications are deployed. The constantly changing target site entails the process to adapt itself. These adaptations include variability in resource selection, extension of the process for error handling and adaptive algorithms for resolving resource dependencies.

In addition, the platform is actively involved in the deployment process. It often needs to participate to the resolution and decision making process to adapt the deployed resources and their configurations. The deployment process should be customizable according to platforms changing requirements and constraints. For instance, the platform

may provide the process with configuration data and influence dependency resolution to fit the underlying system constraints.

- ***Policy customization:*** The ability of the process to allow customization of deployment policies. Custom policies are useful in cases of error and conflict.
- ***Resource type extensibility:*** The ability of the deployment process to be extended for covering different types of resources.

e. Continuous Adaptation

Deployment process adaptation does not only happen during the initial deployment. Throughout the lifetime of the system, adaptations are required such as in [Medvidovic 2007]. Environmental changes may require adapting already installed resources. Newly installed applications may also ask for optimizations or reconfigurations on technical services provided by the platform.

This continuous adaptation process is similar to the autonomic computing loop proposed by [Kephart 2003]. In such paradigm, the deployment agent would be an autonomic manager handling deployment requests, and adapting applications when changes influence the component selection and/or configuration. Notice that pervasive applications are often autonomic [Parashar 2006, Diaconescu 2008] and collaboration between an autonomic manager and a deployment agent is proposed in [Maurel 2010].

- ***Policy based:*** Adaptation logic are developed as code in policies that decides on the deployment actions to take and executes the adaptation process.
- ***Constraint based:*** Adaptations are described as constraints on the platform state. A constraint solver (usually a SAT-based solver) must decide in which state the platform must be, and triggers the deployment process for the suitable change.
- ***Rule based:*** A set of rules constitute the adaptation logic. A rule can be described with conditions on certain platform state or events and the actions to take once these conditions are valid.
- ***Planning based:*** Adaptations are generated as a result of a AI planning algorithm (for example LPG [Kvarnström 2001]) continually working to satisfy given domain knowledge and constraints.

It is important to note that constraint solvers, rule-based systems and planning algorithms that use heuristics do not produce deterministic results for obtaining adaptation logic.

4.3.3 Language Requirements

The third category of requirements includes the criteria on the descriptor language that defines the deployment. The deployment descriptor language is the interface between users (such as developers and operations) and the deployment facilities that conduct the deployment process. It structures and limits the information that can be introduced into the deployment process.

a. Expressivity

The deployment descriptor thus has a pivotal role in the deployment process. Not only it specifies the first deployment of the system but it is also needed for guiding continuous adaptations during its lifetime. Expressivity defines the elements that can be described by the deployment descriptor.

- ***Entities and their relationships:*** The descriptor language allows describing entities to be deployed and the relationships between those, such as dependencies and inclusions.
- ***Platform constraints and requirements:*** The deployment descriptor includes information about the invariants of the deployment platform and the connections between the platform and deployed applications.
- ***Process customizations:*** The deployment language allows to describe custom policies for that the deployment process is customized for a particular instance of deployment.

b. Extensibility

Open environments such as in pervasive computing, deal with a vast heterogeneity of types of entities and artifacts. The deployment of each different type of entity may require different information and procedures. The deployment descriptor language must be able to express a variety of resources, and be extensible with new eventual types. In addition, for a deployment process that is extensible for handling new types of resources, the descriptor language must also describe how the process is extended for handling them.

- ***New entity type:*** The deployment descriptor language allows integrating new entity types.
- ***Inheritance:*** New entity descriptions can be created by inheriting from existing ones, adding other properties.
- ***Substitution:*** The language allows redefining existing entity descriptions according to a well-defined contract [Liskov 1994].

c. Variability

The traditional vision of deployment favors precise description of the system to deploy. Precisely described system deployment does not leave any ambiguities nor choices and leverages determinism of the resulted system. Static, precise deployment descriptions are inadequate to be used for deploying software in dynamic environments. While target environment is unknown and dynamically changing, the deployment descriptor must allow certain amount of variability. The notion of variability is studied in software product lines [Bosch 2002], which is a static, predefined variability. The description of a pervasive deployment must support a higher level of variability, where resource selection, resolution and activation are done at runtime regarding state of dynamically evolving environment. Dynamic product lines address this problem by calculating and deploying new configurations of the system at runtime [Cetina 2008, Parra 2009].

- ***Resource selection:*** The variability described by the descriptor includes information for the adaptation process to choose the changes for applying the variability.
- ***Architectural configuration:*** The deployment descriptor describes different architectural configurations each variability contains.
- ***Dynamic reconfiguration:*** The deployment descriptor language includes information that enables dynamically reconfiguring between different variability choices.

d. Usability

Specifying a deployment is a delicate work, which usually needs high precision and attention. Deployment descriptors are development artifacts, and as any other code must be versioned and tested [Spinellis 2012]. The language of deployment descriptor should be easy to develop in order to simplify the work of platform operators and application developers. Usually reducing the number of concepts would result in a simpler language. However this should not compromise on the expressive power and completeness of the specified program. A better way to quantify this may be the learning curve for new developers and ease of adoption for organizations. Also, as for any code, reuse of already specified descriptors would ease development. The language for deployment descriptor must provide constructs like code inclusion, inheritance or composition.

- ***Declarative language:*** The language describes the structure of the entities and artifacts involved in the described deployment process, instead of expressing the execution flow of the process.
- ***Imperative language:*** The language describes the execution of the deployment process, in terms of sequences of actions to perform.

– ***Language constructs:*** Language structures such as including and referencing other descriptions allow reuse and composition of deployment descriptors.

4.4 Positioning of Related Works

4.4.1 Evaluation of Deployment Platforms

In this section, we position well known platforms and academic works against presented platform requirements and compare them (Table 5.2). Nearly all deployment solutions are built on existing platforms. They enhance standard functionalities on these platforms for providing deployment operations.

Table 4.1: Positioning for deployment platform requirements

Tools	PF.1	PF.2	PF.3	PF.4	PF.5
RPM + Linux	●	●	●	○	○
Puppet + Linux	●	●	●	●	◐
Chef + Linux	●	●	●	○	◐
JVM	○	●	◐	○	○
OSGi™	●	●	●	◐	○
JDrums	●	●	●	○	○
PCOM	●	●	●	●	○
Sofa 2.0	●	●	●	●	○
GatorTech	●	●	●	◐	●
Socam	●	●	●	◐	●
H-omega	●	●	●	●	●

Package managers, such as RPM [Bailey 1997] built on Linux systems, are heavily used in the provisioning of industrial applications. The combination of the underlying operating system and the package manager allow the installation, update and removal of packages dynamically. The package structure, their customization and how dependencies are expressed make them an interesting approach to build Linux-based pervasive systems. With the rise of Cloud Computing, new tools have emerged to ease large-scale deployment [Turnbull 2011, Nelson-Smith 2011]. *Infrastructure-as-code* facilitates creating deployment descriptions. These systems support configuration and reconfiguration of different types of systems. However, they do not support architectural reconfiguration and often rely on packaging systems. Their context management is also limited to predefined data.

The OSGi service platform has become the de-facto modular layer for the Java Virtual Machine. OSGi™ defines a dynamic deployment platform fulfilling most of the platform criteria. With modular deployment capabilities, OSGi constitutes an important foundation for building Java-based deployment platforms. OSGi proposes technics to support architectural reconfiguration by promoting service-orientation. However it involves very

complex code to manage it correctly. In addition it does not provide any context support. Many pervasive platforms, such as H-Omega, GatorTech and SOCAM, are relying on OSGi to deploy applications. They offer a context service responsible for collecting and maintaining contextual data. In addition, H-Omega is based on the Apache Felix iPOJO component model offering dynamic architectural reconfiguration support. But this support is too limited to cover all cases, such as global constraints or contextual bindings.

In academia, early works such as [Andersson 2000] concentrated on defining bases of deployment platforms and stressed importance of modularity and the dynamic update of modules. Later, platforms that provide dynamic reconfigurability feature [Bures 2006, Hoareau 2008] gained focus as foundations for deployment in pervasive environments. Especially, the combination of PCOM [Becker 2004] and BASE [Becker 2003] provides a pervasive platform with architectural reconfiguration capabilities.

By default, all of these systems satisfy configurability and introspection requirements, which is absolutely necessary for any kind of deployment.

4.4.2 Evaluation of Deployment Processes

In this section, we position well-known platforms and academic works against the presented process requirements. Table 4.2 summarizes this study.

Table 4.2: Positioning for deployment process requirements

Tools	PP1	PP2	PP3	PP4	PP5
RPM + Linux	Pull	●	●	●	○
Puppet + Linux	●	●	○	●	○
Chef + Linux	●	◐	●	●	○
Java Web Start	Pull	●	○	○	○
OSGi™	Pull	◐	○	○	○
OSGi™+ Deployment Admin	Pull	●	●	●	○
OSGi™+ OBR / P2	Pull	○	○	●	○
OSGi™+ Apache Ace	Push	●	●	●	○
Software Dock	●	●	○	●	○
OMG D&C	Push	○	○	●	○
Nix	●	●	●	○	○
PCOM	Pull	○	○	●	●
Planning-based	Push	○	○	○	●
Constraint-based	Pull	○	○	●	◐

Package managers enhancing the operating system are providing very customizable transactional deployment processes. Every module can extend the process with ***pre-*** and ***post-*** actions. Unfortunately, they need to be extended in order to support external push. In addition, they do not support any continuous adaptation.

Tools like Puppet or Chef generally adopt a centralized ***master*** server, triggering deployment on remote targets. These targets can impact the deployment process, such as the resource selection. Thanks to the resource-state model promoted by Puppet, it also supports idempotence. However this feature makes the usage of Puppet much more complex for administrators, requiring to shift their mind to this new descriptive model.

Many tools rely on OSGi to enhance their deployment capabilities. OBR and P2 are proposing advanced provisioning functionalities on the top of OSGi. They extend standard OSGi deployment features with advanced dependency management and constraint solving. Deployment admin specification provides a transactional deployment model. Apache Ace is based on the deployment admin and allows deployments from a remote server. However, they do not provide enough flexibility to support continuous adaptations needed for pervasive deployment.

OMG D&C [Object Management Group 2006b] specification defines rigorous principles, actors and actions included in a standard deployment process. It specifies a push model, where released software is configured on target platforms according to a deployment plan. Software Docks [Hall 1999] proposes a deployment agent supporting a very customizable process. It can adapt deployed components to the current environment, and install additional components according to the current constraints. Unfortunately, they do not support continuous adaptation, and do not natively provide a dynamic deployment platform.

Several projects have proposed autonomic deployment process such as ADME [Dearle 2004] and j-ASD [Hoareau 2008]. These approaches are based on constraint-solving to select the components to install. Other projects such as PlanIt [Arshad 2001] and PLASMA [Tajalli 2010] use planning algorithms for calculating deployment plans. PCOM [Becker 2004] also applies adaptation on architecture level for a customizable, continuous deployment process. However, any of these approaches support transactions and their support of the continuous adaptation is not deterministic.

Lastly, Nix [van der Burg 2011] stands apart from other projects as it stresses the importance of transactional deployments. However, it renounces dynamic adaptations in order to provide safer transactions.

4.4.3 Evaluation on Deployment Descriptors

In this section, we position deployment descriptor languages employed by deployment tools against presented requirements. Table 4.3 outlines this positioning. Note that we did not included deployment script languages in our study.

As an early approach to deliver Java applications to Internet clients, Java Web Start provides an XML-based description language, is relatively accessible to developers. However, it doesn't provide any variability over the resources, nor let users express requirements for applications. Puppet and Chef provide domain-specific languages (DSL) for

Table 4.3: Positioning for deployment descriptor requirements

Tools	PD1	PD2	PD3	PD4
Puppet + Linux	●	●	◐	●
Chef + Linux	●	●	◐	●
Java Web Start	◐	○	○	●
OSGi™ + OBR / P2	○	●	●	○
Software Dock	●	●	◐	○
OMG D&C	●	●	○	○
Nix	○	●	●	○
PCOM	◐	●	○	○
Planning-based	●	●	◐	○
Constraint-based	◐	○	●	○

declaring expected system. These languages are based on resources found on operating systems extensible with new types of resources. They provide very little, predefined variability. OSGi provisioning systems, OBR and P2, are based on generic resource model with capabilities and requirements. There is usually a high barrier between these systems and users in terms of usability.

In academia, the description language provided by Software Dock [Hall 1999] is capable of expressing all necessary aspects such as ***constraints***, ***artifacts***, ***dependencies***, ***configurations*** and ***activities***. It lets describing software ***families***, providing a static variability to the description. Projects that use planning algorithms [Arshad 2001, Tajalli 2010] take as input 'facts' such as resource descriptions and constraints. These descriptions are based on generic, extensible models. They let expressing variability but those are hidden implicitly in constraints. Constraint-based approaches [Dearle 2004, Hoareau 2008] provide less extensibility but more explicit variability. Developing and maintaining constraints and rules are difficult. Nix [van der Burg 2011] provides a functional language to describe deployments. Functional programming is useful for the implementing an idempotent deployment process, but it is difficult for developers to debug their descriptions.

4.5 Conclusion

This chapter presents the ideas behind the recent trend of continuous deployment; in which every change created by the software producers is deployed into consumer environments. Continuous deployment is an end result of applying Lean manufacturing principles into software development. The goal of Lean is to create value-added products by less work. For this, it focuses on optimizing the production process in order to eliminate the waste, i.e. the work that do not make it to consumers.

Lean principles are applied in the software development through the concept of deployment pipeline. The deployment pipeline is a pull-based system that automates certain activities of software lifecycle, such as build, tests and deployment. During the process, it ensures that each change included in the development is traced and tested before being delivered to the consumers. Deployment pipeline also implements error reporting and feedback loops in every stage of software delivery. This allows software producer teams (developers, operations, etc.) to work empirically, seeing real repercussions of their effect on the system. Once the deployment pipeline is in place, releasing versions of the software and deploying continuously depends on the automation capacity of the pipeline.

Later, this chapter briefly presents the technologies that are essential to implement the deployment pipeline. Some of these are traditional tools used in software lifecycle, such as version control systems, build automation and artifact management. Others comprise more advanced and recent technologies as continuous integration, deployment automation and testing and finally the monitoring and autonomic control.

The second part of this chapter presents the requirements for enabling continuous deployment, specifically in dynamic execution environments. Similar to the previous chapter's deployment evaluation, requirements are regrouped into three categories, as platform, process and language. Each identified requirement is explained in detail for studying different possibilities in which they are satisfied. Lastly, these requirements serve to evaluate existing work. The result of this evaluation shows which requirements are not yet satisfied by the existing work. ***For deployment platforms***, in spite of some proprietary works, most of the existing platforms lack the capability of providing access to flexible context models. ***For deployment processes***, deployment tools that provide deterministic and fault-tolerant deployment processes do not extend these properties to provide continuous adaptation properties. On the opposite end, works that concentrate on providing continuous adaptation lack on satisfying other requirements. ***For deployment descriptor***, existing deployment tools lack the language support for describing a variable deployment process.

The next chapter presents the proposition of this thesis for addressing a set of well-identified objectives in order to provide continuous deployment facilities for dynamic execution environments.

Chapter 5

Proposition

" When you cease to make a contribution, you begin to die. "

— Eleanor Roosevelt

Contents

5.1 Introduction

The first part of this document introduces the context of this work and presents the state of the art on software deployment in general. Then the previous chapter studies the requirements for the continuous deployment of software systems. This chapter is dedicated to the propositions of this thesis. To begin with, this section starts by spelling out the problems addressed by this research and clarifying the objectives pursued. Then it continues on by presenting the adopted approach and summarizing the overall proposition. After this introduction, different parts of the proposition are presented in detail, together with corresponding discussions. Finally this chapter concludes by evaluating the contributions of this proposition.

5.1.1 Problem Statement

Modern applications raise new challenges for the developers and other stakeholders who participate in the development process. Developing dynamic, scalable software systems that run on distributed and heterogeneous environments goes beyond the realm of programming. From the technical point of view, this requires integrating hardware and software solutions in order to make services available to the use of users. Besides, from the industrial point of view, service and application providers want to maintain these solutions and keep providing users with new and better versions of their services, as fast as possible. Therefore, modern application development is first of all a software engineering challenge, one that requires supplying developers coherent tools and processes to make sure of agile and continuous software delivery.

Considering its particular case, dynamism is one of the requirements and main enablers of this vision. It is a property that is increasingly expected from modern applications. After having been limited to a few domains such as operating systems [Fabry 1976] or pervasive computing [Satyanarayanan 2001]; most recently, dynamism is increasingly expected from modern applications. For instance, applications running on Cloud environments have access to dynamically allocated computing resources that can change at anytime. Similarly, as a next step of modularization of software systems, so-called *traditional* enterprise application providers start to show interest to dynamism. Dynamism allows to handle asynchronous evolution of individual modules separately and without disrupting provided services.

The need for dynamic evolution of modern applications stems from different sources. Three sources that trigger the change can be distinguished [Escoffier 2008]. The first type of change is initialized in a controlled manner, externally and consciously by administrator or configuration of user. Usually, this kind of change is anticipated and planned in advance. The second type of change is due to a decision taken by the application itself. This kind of change is seen in self-adaptive applications. It can be triggered due to a self-optimization or reaction to a change in the internal context of the application, such as

detection of an error. And the third sort of change is due to events external to the application, originating from the execution environment or the surrounding context. This kind of change is usually unanticipated, i.e. happens without the knowledge of the users and the administrators. Nonetheless unexpected things happen very often due to external conditions, such as network disruption or hardware fail. After such change, it is difficult to verify the integrity of the application and decide on which state it supposed to have.

In parallel, it is also important to recognize the administration aspect of these execution environments, regarding the ways software are being delivered to those. The recent years have witnessed the proliferation of application platforms. In not so distant past, only the operating systems of personal computers allowed users to easily install applications. Now different domains, such as Cloud, mobile and more recently pervasive environments, see stakeholders providing execution platforms and application stores that allow delivery and execution of applications easily over Internet. On the system administration front, this creates a duality over type and control over the software management. On one hand, the platform providers need to make sure that the software platform that provides support (APIs, access to resources, etc.) for applications is working as expected. This requires exhaustive testing of those platforms, against many scenarios and thus relatively little and slow evolution of the software. On the other hand, the application developers want to continue reaching users in different conditions, but also push new functionalities as early as possible. So the applications need evolving more rapidly and dynamically to the changes. As a consequence the development process of applications running in dynamic environments is hindered by the lack of tools that rapidly and automatically reproduce runtime environments used for testing, production, etc.

In this picture, the deployment process gains importance for the software delivery, as it is the crucial step that transforms passive code into an active entity. However, despite many in-depth studies on this domain, traditional approaches fail to address the above-mentioned challenges of dynamic systems. Often reduced to a process executed once and for all, current deployment solutions need to be extended for dynamic environments. The recently emerging field of continuous deployment is a promising candidate for responding to the deployment needs of dynamic environments. It incorporates a set of practices aiming to provide a process for deploying software rapidly and predictably; whether for the first provisioning of the system or for the adaptation of the running applications. The previous chapter studies requirements for enabling continuous deployment on dynamic environments and concludes that current deployment facilities fall short on satisfying those. For this kind of environments and applications the continuous deployment features need to take into account different requirements of the execution platform and applications, updates of separate modules, as well as their reconfigurations to cope with the evolving context.

5.1.2 Research Objectives

The main goal of this research is to ***enable continuous deployment on dynamic execution environments***, so that software systems continuously get updated and adapt to the changes of their internal configurations and external context. Particularly, this thesis proposes a set of deployment facilities that are adequate for reaching this goal and demonstrate how these facilities can be implemented and used. These facilities comprise;

- the ***process definition*** that allows continuous deployments,
- the ***reference architecture*** for a deployment manager and
- the ***domain-specific language*** for describing deployment tasks.

The requirements for achieving continuous deployment are already listed and explained in the chapter 4. In addition to the satisfaction of those requirements, the contribution of this thesis pursues four major objectives for the proposed solution and discusses their implications along this chapter. These objectives are: ***reproducibility*** and ***fault-tolerance*** of the deployment process, providing means for ***continuous adaptation*** and the ***tooling support*** for the proposed facilities. Each objective address a number of underlying challenges, as shown in the Table 5.1.

Table 5.1: Research objectives and addressed challenges

Objectives	Challenges
Reproducibility	Scalability, Heterogeneity
Fault-tolerance	Distribution, Industrialization
Continuous Adaptation	Dynamism, Context-awareness
Tooling	Automatization, Testability

Reproducibility — For build systems that produce executable software artifacts from the source code, reproducibility is a common practice: It is expected that every time same source code is built, the resulting artifacts will be the same. As for the deployment facilities, they produce executing software from software artifacts. Similarly a deployment process must ensure that repeating the same deployment operation in different deployment sites produces the same result on every site. If a deployment facility ensures reproducible process, only then it can be used to deploy on large scale, with many deployment sites. Total reproducibility is difficult to achieve because every deployment site is different in its actual state, which is sometimes unobservable. On the upside, it is possible to achieve a partial reproducibility, meaning that the system guarantees that a local, well-defined part of the system configuration is as expected. This, in turn, gives the ability to deploy software on heterogeneous environments.

Fault-tolerance — One of the essential properties of software is the fault-tolerance, which is the ability to continue functioning under the conditions that are not expected. Yet it is seldom possible to provide a full coverage of faults and errors that may occur in real-world systems. Any software system needs to guarantee a minimum of fault-tolerance to be used in industrial scales. Moreover, attaining fault-tolerance is even more challenging in distributed systems, because of the unpredictable nature and scale of underlying network infrastructures. In modern applications coordinating distributed computing elements has became mundanely common. Deployment is a critical process that is liable for failures, because of the fact that it changes the state of possibly already running systems. Because of these reasons, it is even more difficult to coordinate the deployment on multiple machines found in distributed systems. Providing industrial-scale, distributed solutions for deployment depends on the ability of the deployment process to be fault-tolerant.

Continuous Adaptation — As expressed in the beginning of this chapter, the environments targeted in this work are characterized by their unpredictable, dynamic evolution. The successful introduction of a software system in such environments is not the end of the deployment, but it marks the beginning of the runtime management of the deployed system. Managing the system at runtime requires monitoring its state and continuously adapting it through deployment actions such as updates and reconfigurations. Continuous adaptation allows applications to handle dynamic changes. This way, applications can continue to be operational despite unexpected side effects of dynamism. Moreover, such adaptation capabilities are the foremost prerequisite for creating context-aware applications. Context-aware applications can deliberately alter their configuration and behavior according to the current state of their environments.

Tooling — The adoption of a software solution is directly related to its ease of use, the more so when the target domain is complex. The ease of use is reflected through the mechanisms implemented to simplify the work of developers for performing tasks in hand. Tools help reducing the complexity by performing certain tasks automatically. Two important tasks that constitute deployment are the specification of the deployment and the execution of the deployment process according this specification. Deployment facilities are, by definition, about automation of deployment operations. Yet this is, in general, confined to the execution of the deployment process. Specification of the deployment is all so important and can get very complex for even small-sized systems. Easy to use deployment facilities are crucial to increase manageability of the deployment process, therefore that of the deployed system. Another aspect that increases the ease of use of the proposed system is its testability. The management of complex software systems is not possible without tools that allow to test deployment specifications.

5.1.3 Approach

The state of the art on continuous deployment presented in the previous chapter 4 concludes that the existing works fall short on satisfying all the necessary requirements for the deployment process. The contribution of this thesis concentrates, in the first place, on providing a deployment process that satisfies the mentioned requirements. In order to do so, the following definition of the deployment process is used throughout this proposition.

Definition 6: Software Deployment Process

The software deployment process is a coordination of operations that brings a software system from its actual state to a target state.

This definition is chosen explicitly to put emphasis on the coordination and state transition aspects of the deployment process. The first step of this approach, therefore, is to formally define the problem domain of the software deployment process. Three parts can be extracted from this definition namely, (1) *the current state* of a software system, (2) *the target state* expected to be effective on the system and (3) *the transition process* that applies the expected state onto the actual execution environment.

Before proceeding further, it merits noticing that the contribution of this work does not handle distributed deployment. For concentrating on mentioned aspects, it is restricted to the deployment of applications running on a single site. Nevertheless, these applications can, and often does rely on remote services.

The first part of this proposition, presented in section 5.2, consists of providing a formal framework for these concepts. This framework is based on the generic concept of ***resource*** to represent current and target states of the platform. Resource concept allows the deployment process to be extensible for covering different kinds of entities that can be deployed on deployment sites. In addition to this, the target state of the system can be described as a resource graph, this time with different levels of precision and variability. This allows expressing possible configurations of the applications regarding different states of the execution environment. This declarative approach is advocated by the *Infrastructure-as-Code* approach and the *DevOps* movement [Spinellis 2012].

The next step of this approach is to define a deployment process on the top of state descriptions. This process is based on a transformation operation on resource graphs, which provides means to combine multiple descriptions into one. In other words, it describes the coordination of resource state transitions that occur during a deployment process. The section 5.3 outlines how the operation serves to define the deployment of a target description on an execution platform. This operation holds certain formal properties, including ***idempotence***. Later discussions in the section 5.4 argues how the fault-tolerance and reproducibility are granted to the deployment process thanks to these properties.

Then the second part of this proposition, presented in the section 5.5, consists of designing the reference architecture for a deployment manager that implements the formally described deployment process. This architecture aims to complete the promise of the deployment process by providing continuous deployment facilities for dynamic environments. The figure 5.1 shows the overview of this architecture.

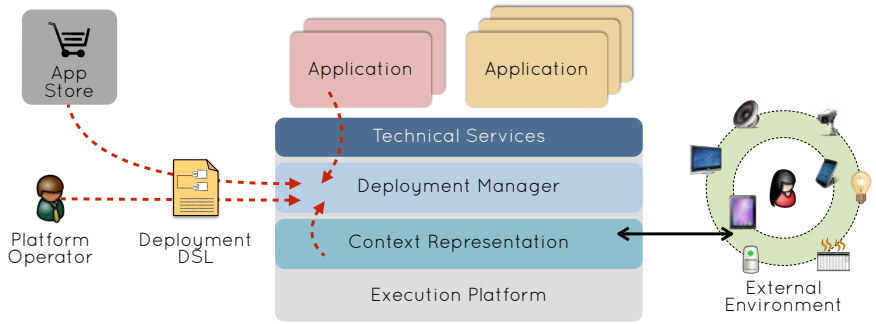


Figure 5.1: Proposition Overview

The context representation is in charge of modeling the current state of the platform. It provides a resource-based representation, on which each represented entity (resource) yields interfaces for observing and manipulating its state. This representation is extensible with different kinds of entities in order to cover possible resource heterogeneity on dynamic environments, notably in pervasive environments.

The deployment manager treats deployment requests triggered from different sources and, if necessary, plans and executes the deployment process. The execution of the deployment process is implemented inside local transactions for ensuring the ***fault-tolerance***. In addition to the implementation of the deployment process, the deployment manager provides management support for applications. Depending on the capabilities of the underlying context representation, it activates monitoring policies for each deployed application. Together with custom adaptation policies, it is capable of continuously adapting managed applications according to the changes on the platform state.

The third and last part of this proposition, presented in the section 5.6, involves the design of a domain-specific language (DSL) for describing the target state, as determined by the formal framework. This language allows to design deployment descriptors for applications containing variability. It constitutes a basis for the ***tool support*** and ensures the ease of use of the proposed deployment facility.

In the remaining sections of this chapter each one of these contributions are presented in detail. Finally the last section of this chapter concludes this proposition by evaluating these contributions in relation with the research objectives.

5.2 Formalization of Deployment Concepts

This section aims to present the conceptual framework, which guides the main propositions of this thesis. The framework formally defines the concepts of software deployment based on the Set Theory and the Graph Theory. The concepts are presented in three groups as; resource related concepts, assembly related concepts and application related concepts. Each concept is defined together with the rules and operations they participate in. Finally, these concepts allow to describe the algorithms used during the deployment process.

5.2.1 Resource Related Concepts

The ***resource*** is the elementary concept manipulated during the deployment process. A resource describes an entity state. This entity is intended to be found on a deployment site in the given state. This way a deployment site is composed of a set of resources, which represent all the relevant information of the environment. Resources contain sufficient information to represent the knowledge of state and configuration of each entity. The state of the entity is modeled by a set of ***properties***.

a. Property

The first concept of the formalization is the ***property***. Let k a name and v a value, then property p is defined as a pair of name and value (See equation 5.1).

$$\begin{aligned} p = (k, v) \mid k \in \mathbb{K} \quad \text{and} \quad v \in \mathbb{V} \\ \mathbb{K} = \{\text{Set of all valid property names}\}, \\ \mathbb{V} = \{\text{Set of all valid values}\} \end{aligned} \tag{5.1}$$

Notice that here the value type of properties is omitted in the definition for the sake of simplicity. One can simply extend the definition of property by defining data types and valid values for each types, such as *Boolean*, *Integer*, *String*, etc. The real set containing $\mathbb{K}$ is also simplified for the same reason.

b. Resource Type

The generic concept of resource covers a large spectrum of entities like services, files, executable binaries, their configuration, code libraries like Dll and Jar files, hardware resources like network ports, disk space, available memory, available peripherals etc. To be able to manage this heterogeneity different types of resources are defined by the ***resource type*** concept.

The ***resource type*** determines the set of property names that are valid for resource descriptions. Each resource type identifies three sets of property names that are used to

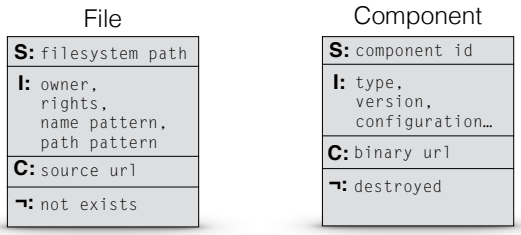


Figure 5.2: Resource Type Examples

describe resources belonging to that type[1]. Different kinds of properties serve to identify the resource description level, which is discussed below. In addition to that, each resource type defines a special property that describes the state of absence, called *negative property*, which is also discussed below in detail. And finally a function f that prescribes the operations to be performed on the deployment site for changing resource state. The internal modeling of the state changing functions is out of the scope of this work. To give an idea, they can be thought as functions that take initial state and target state as resource descriptions and perform a set of operations. An example signature for a state transition function would be $f = (r_{initial}, r_{target})$.

Let I is a finite set of property names called ***Inquiring***, S is a subset of I defining property names called ***Specifying***, C is a finite set of property names defining property names called ***Constructing***, $\neg$ the negative state property and the function f the state transition function. Then the resource type t is defined with the following equation 5.2.

$$\begin{aligned} t = (I, C, S, \neg, f) \mid S \subseteq I \text{ and } C \cap I = \emptyset \\ I, C = \{\text{Finite set of property names }\}, \\ \neg = (k, v) \mid k \in I \end{aligned} \tag{5.2}$$

Notice that including valid property values for each property name defined by the resource type can enhance this definition This is also left out in this formalization because is not the core of this work.

c. Resource

The ***resource*** concept delineates the state of entities found on deployment sites. According to the type of the resource, the deployment process applies different actions for bringing resources to their intended states. For example, for a resource of type ***file***, the deployment process should first identify the hypothetical file described by the resource and then make sure that the actual file exists on given location, possessing described properties and content. If the file does not already comply with the described state, the deployment process should use the information given by the resource description (file path, file source, content, permissions, etc.) in order to apply necessary actions for creating or updating the file. On the other hand, there are resources, like available memory,

[1]These set of property names do not contain duplicate names.

that the deployment process does not have direct access on. These types of resources are still relevant for the deployment but all the deployment process can do is to check if an entity exists with the state described by the resource.

A resource is defined with its type and the set of properties that describes its state. Let t is a resource type and P is a finite set of properties, the resource r is defined as the pair type and properties (See equation 5.3).

$$\begin{aligned} r = (t, P) \mid t = (I, C, S, \neg, f) \in \mathbb{T},\ \forall p \in P \mid p = (k, v),\ k \in (C \cup I) \\ \mathbb{T} = \{\text{Set of resource types}\}, \\ P = \{\text{Finite set of properties}\} \end{aligned} \tag{5.3}$$

An alternative but equivalent definition of the resource concept would be defining the properties into separate sets according to different types of properties (See equation 5.4).

$$\begin{aligned} r = (t, P_I, P_C, P_S) \mid t = (I, C, S, \neg, f) \in \mathbb{T}, \\ \forall p \in P_I \mid\ p = (k, v), k \in I, \\ \forall p \in P_C \mid\ p = (k, v), k \in C, \\ \forall p \in P_S \mid\ p = (k, v), k \in S, \end{aligned} \tag{5.4}$$

Universal set of all possible resources is noted as $\mathcal{R}$.

$$\mathcal{R} = \{\text{Set of all resources}\} \tag{5.5}$$

For a given resource r, its type is noted $typeOf(r)$ and its property set is noted $propertiesOf(r)$.

$$\forall x \in \mathcal{R},\ x = (t_x, P_x) \mid\ t_x = typeOf(x) \text{ and } P_x = propertiesOf(x) \tag{5.6}$$

d. Resource Description Levels

As defined by the resource type, properties included in a resource description can be inquiring, specifying or constructive. This gives the possibility to describe resource state in various levels, conforming to the type it belongs. Three rules are defined on the resource concept that decides the level of description of resources. These rules give rise to four subsets of the universal resource set, noted $\mathbb{S}$, $\mathbb{I}$, $\mathbb{C}$ and $\mathcal{N}$.

Specified resource refers to a resource description that includes necessary information to be matched with a ***specific entity***. For example, given a file resource type, a specified resource should include the file's canonical path to be able to point to the exact file on the deployment site's filesystem. Let resource $r = (t, P)$ with resource type $t = (I, C, S)$, r is called ***specified resource*** if all the specifying property names defined by the t are included in the properties of r. A specified resource is thus quantified as shown in the equation 5.7.

$$\begin{aligned} \forall r \in \mathbb{S} \mid r = (t, P),\ t = (I, C, S, \neg, f) \\ \forall k \in S,\ \exists p \in P \mid p = (k, v) \end{aligned} \tag{5.7}$$

Constructive resource refers to a resource description that includes enough information to construct the given resource in case it is necessary for the deployment process. Following the example on file resource type, a constructive resource would include the source identifier of the file or the content, along with the path that is intended to be placed on the filesystem. Notice that some resource types may not define any constructive property names, thus it is not possible to construct resources of that type. An example for these resources are hardware interfaces. A deployment system can check if a hardware interface, a USB device for instance, is present on the system but cannot construct it. Let resource $r = (t, P)$ with resource type $t = (I, C, S)$, r is called ***constructive resource*** if all the constructive property names defined by the t are included in the properties of r. It can be quantified as in the equation 5.8.

$$\begin{aligned} \forall r \in \mathbb{C} \mid r = (t, P),\ t = (I, C, S, \neg, f) \\ \forall k \in C,\ \exists p \in P \mid p = (k, v) \end{aligned} \tag{5.8}$$

Inquiring resource refers to a resource description that neither specified nor constructive, but describes the state of an hypothetical resource entity that exist on the deployment site. Instead of referring to a particular resource entity, inquiring resources describe a query on the deployment site, whether such resource described by the properties exists. An inquiring resource of type file, for example, may describe a file that is located under a particular directory, with a particular file extension, owner or access rights. This knowledge is not enough to create the file if it does not exist, or check its exact state, such as its content. Let resource $r = (t, P)$ with resource type $t = (I, C, S)$, r is called ***inquiry resource*** if all properties of resource r are inquiry property names defined in t. It can be quantified as shown in the equation 5.9.

$$\begin{aligned} \forall r \in \mathbb{I} \mid r = (t, P),\ t = (I, C, S, \neg, f) \\ \forall p \in P,\ \exists k \in I \mid p = (k, v) \end{aligned} \tag{5.9}$$

Negative resource refers to a resource description that describes the ***absence*** of the described entity. Each resource type defines a negative property $\neg$ that describes a state of the resource in which it ***does not*** exist on the deployment site. For a resource type file, it can be the state *"deleted"* or simply *"absent"*. Notice that the value of this property is usually a *String* or a *Boolean* that represents the non-presence semantic depending on the resource type. For example, a system service's negative property can be defined either as *"not present"* or *"not started"*. A negative resource can be quantified by the equation 5.10.

$$\begin{aligned} \forall r \in \mathcal{N} \mid r = (t, P), t = (I, C, S, \neg, f) \\ \exists p \in P \mid p = \neg \end{aligned} \tag{5.10}$$

It is possible for a resource state to have mixed description levels. For example, a specified resource can include a number of inquiry properties that narrows down and describes more precisely the state of the resource. Constructive resources can also include inquiring and specifying properties to describe and specify the resource to be constructed. Relationships between different description levels are illustrated in the schema 5.3. To

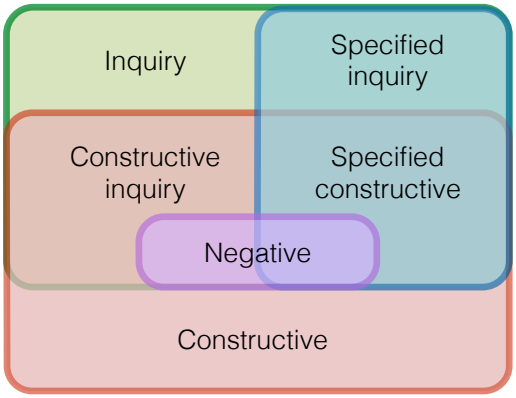


Figure 5.3: Resource Description Levels

summarize resource description levels, inquiry properties comprise the description of the expected state of the resource; specifying properties hold the information for pointing to a specific entity and constructive properties hold the information whether the resource will be constructed, if yes how. And these three types of properties can be used together in a resource description (see Figure 5.4). In parallel to these three description levels, a negative resource designates the state of absence of the described resource.

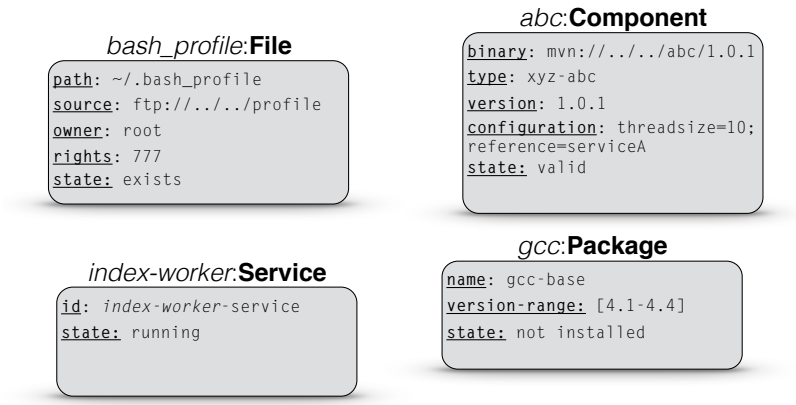


Figure 5.4: Resource Examples

e. Resource Comparison

Based on resource properties and description levels, it is possible to compare resources. Following rules lay out concepts of equivalence, inclusion and conflict between resources.

Resource Equivalence Two resources are said ***equivalent*** if they belong to the same type and their property set are the same, i,e. they have exactly the same properties, including name and values. In order words, let r_1 and r_2 two resources. r_1 and r_2 are equivalent if their type and property sets are equal (See equation 5.11). Equivalence between r_1 and r_2 would be noted as $r_1 \equiv r_2$.

$$\begin{aligned} &\forall r_1, r_2 \in \mathcal{R} \\ &\quad typeOf(r_1) = typeOf(r_2) \\ &\quad propertiesOf(r_1) = propertiesOf(r_2) \Leftrightarrow r_1 \equiv r_2 \end{aligned} \tag{5.11}$$

Resource Subsumption A resource ***is subsumed into*** another if they are the same type and one's properties completely includes the other's. Let r_1 and r_2 two resources. r_1 is subsumed into r_2 if they are the same type and the property set $P(r_1)$ of r_1 is a subset of property set $P(r_2)$ of r_2 (See equation 5.12). Subsumption between r_1 and r_2 is noted as $r_1 \sqsupseteq r_2$, or $r_2 \sqsubseteq r_1$.

$$\begin{aligned} &\forall r_1, r_2 \in \mathcal{R} \\ &\quad t(r_1) = t(r_2) \\ &\quad P(r_1) \subseteq P(r_2) \Leftrightarrow r_1 \sqsupseteq r_2 \end{aligned} \tag{5.12}$$

If two resources are equivalent, they are subsumed into each other.

$$r_1 \equiv r_2 \Leftrightarrow r_1 \sqsubseteq r_2 \text{ and } r_1 \sqsupseteq r_2 \tag{5.13}$$

Resource Conflict Two resources are in ***conflict*** if they refer to the same specific resource entity but describe different states via their properties. Let r_1 and r_2 two resources. r_1 and r_2 are conflicting if their type and their specifying properties are equal, they do not subsume one another and their entire set of properties are not equal (See equation 5.14). Conflict between r_1 and r_2 would be noted as $r_1 \bowtie r_2$.

$$\begin{aligned} &\forall r_1, r_2 \in \mathcal{R} \\ &\quad t(r_1) = t(r_2) = (I, C, S, \neg, f) \\ &\quad \forall k \in S\,,\ \exists p = (k, v) \mid p \in P(r_1)\,, p \in P(r_2) \\ &\quad r_1 \not\sqsubseteq r_2\,,\ r_1 \not\sqsupseteq r_2 P(r_1)\,, \neq P(r_2) \Leftrightarrow r_1 \bowtie r_2 \end{aligned} \tag{5.14}$$

5.2.2 Assembly Related Concepts

Resources are regrouped together inside ***assembly*** structures. Inside an assembly, relationships can be expressed between resources. Assembly structures allow to bring resources that have associations relevant to the context of deployment.

a. Dependency

Naturally resources depend on each other to be able to function properly. A basic example would be a software component requiring a file to read and write data. In this case the component depends on the file. Dependency is a kind of *"use"* relationship between two resources. Resources can use, thus depend on, one another in various stages of software life-cycle, such as development time, deployment time and finally at runtime. In the previous example (figure 5.4), although the component depends on the file, the absence of the file probably won't pose any problems during the installation of the component. However, at execution, the absence of the file may have an effect on the operation of the component. In return a software component can also depend on another component, which it requires during deployment and execution. The concept of dependency is not equivalent nor do cover the notion of ***binding***, a term that usually refers to the runtime references between software components. Instead, the dependency relationship models the requirements of a particular resource to attain the state described by its properties.

Let R is a finite set of resources, dependency is an asymmetric, intransitive binary relation in R. For $r_1, r_2 \in R$ dependency relation would be noted $r_1 \rightarrow r_2$. It is said that r_1 ***depends on*** r_2.

Figure 5.5: Resource Dependency

It is important to note that the dependency concept here represents only mandatory requirements, as opposed to the concept of ***optional*** dependencies. The concept of optional dependency appears in many dependency systems. It describes the cases where an entity ***will continue*** to operate in the absence of its optional dependencies but in a different manner.

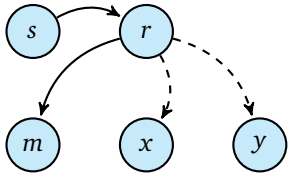


Figure 5.6: Optional Dependency System

In this formalization framework such a special case is not needed because resources describe the state of entities. Indeed, an entity having an optional dependency can be represented as two resources with different states, representing the same entity, and one having an additional dependency for the case of optional dependency. There are pros and cons of such a modeling preference. On the downside, representing entities having multiple optional dependencies is difficult. This will cause as much different resources as there are combinations of availability of optional dependencies. On the upside, the resources represent entities in a precise manner. Each case of availability and unavailability of an

optional dependency is represented with a resource state. Representing an entity with two optional dependencies (noted here with $\rightsquigarrow$) would be like the following.

Let r, s, m, x and y are resources

$$\begin{array}{l} s \to r \\ r \to m \\ r \rightsquigarrow x \\ r \rightsquigarrow y \end{array} \Rightarrow \begin{array}{l} s \to r_\emptyset \\ r_\emptyset \to m \end{array} \;\&\; \begin{array}{l} s \to r_x \\ r_x \to m \\ r_x \to x \end{array} \;\&\; \begin{array}{l} s \to r_y \\ r_y \to m \\ r_y \to y \end{array} \;\&\; \begin{array}{l} s \to r_{xy} \\ r_{xy} \to m \\ r_{xy} \to x \\ r_{xy} \to y \end{array} \tag{5.15}$$

Graph Theory

A graph G is defined as a pair of sets $G = (V, E)$, where V is the set of ***vertices*** and E is the set of ***edges***, formed by pair of vertices.

An alternative definition defines explicitly the direction of the edges is $G = (V, E, s, t)$, where $s, t : E \to V$ are two relations indicating source and target vertices of edges. If sources and targets of edges are differentiated, these graphs are called ***directed graphs***, noted $\mathcal{G}^d$.

A graph is called ***simple*** if there is no loops and no multiple edges between two distinct vertices. A graph is ***connected*** if there is a path between every pair of vertices in the graph. If a graph is not connected, than each subset of vertices that are connected with each other with edges are called ***components*** (not to be confused with software components).

Directed Acyclic Graph or a DAG is a directed graph with no directed cycles, noted $\mathcal{G}_a^d$. That is, when traversing vertexes in the direction of edges, there is no way of passing from the same vertex a second time.

b. Assembly

The highly generic concept of resource and simple dependencies leave many possibilities to express complicated dependency systems. A resource, for example a software component, can express its requirements for other software components and services, which are another resources; on a file, which is another resource; requirement on a container property, and yet on a feature of the operating system, which can also be described as resources.

The concept of ***assembly*** models a dependency system with a set of resources and dependency relations between those. Let R a finite set of resources and D a dependency relation defined on top of R, an assembly is defined as in the equation 5.16.

$$A = (R, D) \mid R \subset \mathcal{R},\ D \subseteq R \times R. \tag{5.16}$$

This assembly definition is equivalent to that of a directed graph, with vertices as resources and edges as dependencies. Resources as assembly vertices already contain properties. Indeed a more specific equivalence would be to the concept of ***property graph***, in which each vertex and edge are associated with a set of key/value properties, as defined in the concept of property.

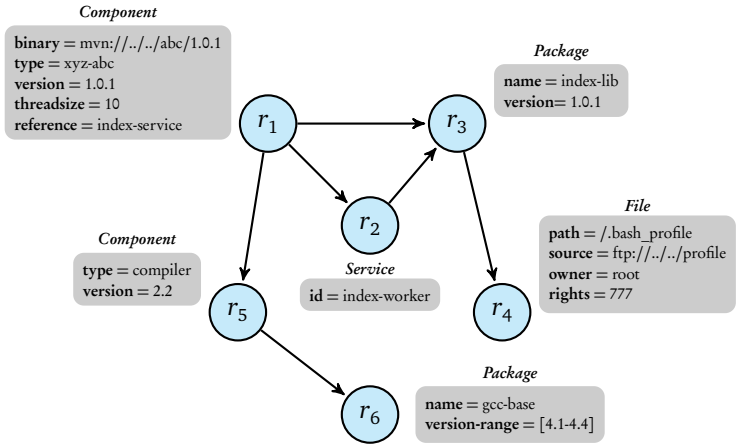


Figure 5.7: Assembly Example

The universal set of all possible assemblies is noted as $\mathbb{A}$. In its definition, a graph does not exist with an empty set of vertices. For this, an empty assembly is defined as ε, as a theoretical assembly without resources and dependencies, defined in equation 5.17.

$$\begin{aligned} &\varepsilon = (R, D) \mid R, D = \emptyset \\ &\mathbb{A} = \{\, \text{Set of all assemblies} \,\} \\ &\mathbb{A} = \mathcal{G}^d \cup \{\, \varepsilon \,\} \end{aligned} \tag{5.17}$$

For an assembly A, the set of resources it contains is noted $resourcesOf(A)$ and the set containing dependency relations between these resources is noted as $dependenciesOf(A)$ (See equation 5.18).

$$\forall M \in \mathbb{A}, M = (R_M, D_M) \mid R_M = resourcesOf(M) \quad \text{and} \quad D_M = dependenciesOf(M) \tag{5.18}$$

Assembly Validity An assembly is called ***"valid"*** if there is no conflict between its resources and dependencies between resources does not form a cycle. The set of valid assemblies is noted as $\mathbb{A}^*$. Given the acyclic nature of the dependency relations in valid assemblies, the graph representing a valid assembly is a directed acyclic graph or a DAG, noted $\mathcal{G}_a^d$. Assembly validity is defined in equation 5.19.

$$\begin{aligned} &\forall A \in \mathbb{A}^* \mid A \in \mathcal{G}_a^d, \nexists r_1, r_2 \mid r_1 \bowtie r_2, \text{ where} \\ &\quad \mathbb{A}^* = \{\, \text{Set of all valid assemblies} \,\} \\ &\quad \mathbb{A}^* \subset \mathcal{G}_a^d \end{aligned} \tag{5.19}$$

An assembly is defined non-valid when it contains dependency cycles or resource conflicts. However, when an assembly contains resources that are equivalent with each other, or that subsumes one another, it is still deemed valid. This is because such assemblies can be reduced by eliminating resources that are equivalent or that subsume each other; without loosing information.

Assembly Completeness An assembly is called ***"complete"*** if it is valid and for each inquiry resource of the assembly is found at least one specific resource that subsumes the inquiry resource. It is said that all inquiry resources are resolved inside the assembly. The set of complete assemblies is noted as $\mathbb{A}^+$. Note that although all complete assemblies are valid by definition, a valid assembly is not necessarily complete. Assembly completeness is defined in the following equation 5.20.

$$\forall A \in \mathbb{A}^+ \mid A \in \mathbb{A}^*,\ r_1, r_2 \in resourcesOf(A), \forall r_1 \in \mathbb{I}, \exists r_2 \in \mathbb{S} \mid\ r_2 \sqsubseteq r_1 \quad (5.20)$$

The above remark about reducing assemblies is also viable for complete assemblies. Eliminating subsumed inquiry resources such that all remaining resources are specified or constructive can reduce a complete assembly. Such operations on assemblies are defined in the following section.

Following are a suite of concepts that are linked to the graph theory and graph transformations. These theories are used to define the deployment process and the algorithms that can be used for coordinating this process. The properties shown by means of these theories are later discussed and serve to discuss that the approach presented in this thesis meets its objectives.

Graph Operations

Graph operations produce new graphs from input graphs. The changes to a graph as a result of an operation is represented using ***graph morphisms***. Let $G = (V_H, E_H, s_H, t_H)$ and $H = (V_H, E_H, s_H, t_H)$ are two graphs, a graph morphism is defined by a function $f : G \to H$, $f = (f_V, f_E)$, which consists of two functions $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ such that f_E and f_V are compatible with source and target mappings, with $f_V \circ s_G = s_H \circ f_E$ and $f_V \circ t_G = t_H \circ f_E$.

The composition of two morphims $g = (g_V, g_E)$ and $f = (f_V, f_E)$ is noted as $g \circ f = (g_V \circ f_V, g_E \circ f_E) : G \to I$ with $f : G \to H$ and $g : H \to I$.

Using graph morphism, it is straightforward to define simple, local changes to graphs, such as addition or deletion of vertexes and edges.

A useful elementary operation is vertex contraction (identification), where two vertices of a graph are identified and merged into one vertex, assembling the edges into that merged vertex. Let $G = (V, E, s, t)$ is a graph, vertices $x_1, x_2 \in V$, then it is noted $contraction(G, x_1, x_2) : G \to G'$.

The vertex contraction operation is used in the construction of ***quotient graphs***. A quotient graph is constructed from a graph G with vertex set V and an equivalence relation $\sim$: $V \to V$ such that all relations in $\sim$ are identified and contracted in G. Quotient graph of G according to the equivalence class $\sim$ is noted $G_{/\sim}$.

A common binary operation is the ***disjoint union*** of two graphs. For two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, their disjoint union is $G \oplus H = (V_G \cup V_H, E_G \cup E_H)$.

Using quotient graphs and disjoint union, it is possible to write another binary operation which ***glues*** two graphs to each other. Gluing of two graphs G and H is simply the quotient graph of the disjoint union of two graphs, $(G \oplus H)_{/\sim}$. Gluing constructions of graphs gave rise to the domain of ***graph transformations*** by the seminal work of [Ehrig 1973].

c. Unary Operations on Assemblies

The fact that assemblies are defined as graphs allows to use graph operations and graph theory methods to analyze and manipulate assemblies. This section defines two operations, resource identification and resource subsumption built upon already existing operations on graphs.

Resource Identification is an unary operation defined on assemblies. It manipulates the given assembly, contracting resources that are equivalent. The resulting

assembly does not have any resources that are equivalent to each other. Resource identification over an assembly A is noted as $identify(A)$. It is indeed a quotient graph of A over the equivalence relation $\equiv$, which can be noted $A_{/\equiv}$. An algorithmic definition of the operation would be like the following.

Procedure identify(assembly A)

Input: $A = (V, E) \in \mathbb{A}$.
Result: $A = (V', E') \in \mathbb{A}$ with $V' \subseteq V$ such that $\nexists r_1, r_2 \in V' \mid r_1 \equiv r_2$

```
1 forall the r1, r2 ∈ A | r1 ≡ r2 do
2     A ← contraction(A, r1, r2)        // contract equivalent
3 return A
```

- **Non-closure** Let assembly $A = (R, D) \in \mathbb{A}^*$, resource identification operation is not closed on the set of valid assemblies. Identification operation can result in cycles in the dependencies (Equation 5.21).

$$\exists A \in \mathbb{A}^* \mid \texttt{identify}(A) \notin \mathcal{G}_a^d \tag{5.21}$$

- **Idempotence** Let assembly $A = (R, D)$, resource contraction operation is idempotent. Applying resource contraction multiple times has no effect on the resulting assembly (Equation 5.22).

$$\forall A \in \mathcal{G}_a^d, \texttt{identify}(\texttt{identify}(A)) = \texttt{identify}(A) \tag{5.22}$$

Resource Subsumption is another unary operation defined on assemblies. Given an assembly, it merges resources that subsume each other for eliminating multiple declaration of a ***specified resource***. Resource descriptions and assemblies describe final expected state of resource entities. An assembly containing two resources, one subsuming another, means that same specific resource entity is described with two different levels of detail. Therefore this information is redundant. Resource subsumption eliminates this redundancy by merging subsumed resources into the upper resource, gathering dependencies into that upper resource. The algorithmic description of this procedure is defined as the following.

Procedure subsume(assembly A)

Input: $A = (V, E) \in \mathbb{A}$.
Result: $A = (V', E') \in \mathbb{A}$ with $V' \subseteq V$ such that $\nexists r_1, r_2 \in V' \mid r_1 \sqsubseteq r_2$

```
1 forall the r1, r2 ∈ A | r1 ⊑ r2 , r1 ≢ r2 do
2     A ← contraction(A, r1, r2)        // contract subsuming
3 return A
```

Like the resource identification operation, resource subsumption is also non-closed on the set of valid assemblies and idempotent.

Graph Transformation

Graph transformation or graph rewriting consists of techniques that aim to create a new graph out of an original graph algorithmically. There are different approaches for transforming graphs, such as algebraic-categorical approach, term graph rewriting and matrix graph grammars. In the context of this work, algebraic-categorical approach is presented in more details.

The gluing construction of two graphs was defined previously in a set-theoretical way, using quotient graphs, in the form $(A \oplus B)_{/\equiv}$. Starting from the gluing construction algebraic approach for graph grammars aims to generalize the notion of Chomsky grammars for constructing graph grammars that apply transformations [Ehrig 1979]. Like for Chomsky grammars, a graph transformation system is based on ***production rules*** that describes the kind of change that will transform certain graphs into others. For the case of graph grammars a production rule describes graph morphisms. In the double-pushout approach (DPO), a production rule, or a rewriting rule, $r = (L \leftarrow K \rightarrow R)$ consists of two graph morphisms, where $K \rightarrow L$ is ***injective***. Briefly explained, in DPO a production rule describes the part of the graph to be deleted (the left-hand side $K \rightarrow L$) and the part of the graph to be inserted (the right-hand side $K \rightarrow R$). The application of the rule r on the initial graph G to form the target graph H is noted $G \Longrightarrow_r H$.

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ \downarrow & & \downarrow & & \downarrow \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

$G \Longrightarrow_r H$ is called a direct derivation via r, based on $K \rightarrow L$. A derivation $G \Longrightarrow_* H$ means G is transformed into H as a result of a sequence of finite direct derivations $G = G_0 \Longrightarrow_{p_1} G_1 \Longrightarrow_{p_2} \cdots \Longrightarrow_{p_{n-1}} G_n = H$.

d. Monoid on Assemblies

In this section a monoid structure on the set of assemblies is defined. A monoid is an algebraic structure with a single binary operation that is ***associative*** and that has an ***identity element***. This section presents the ***join*** operation and discusses its properties.

The assembly monoid is defined over the binary operation **assembly join**. This operation aims to glue two ***valid*** assemblies all by preserving the validity of the resulting assembly. The ***join*** of two valid assemblies A and B is noted **A@B**. As discussed previously, a gluing operation consists of applying a disjoint union of two graphs and then contracting the resulting graph according to an equivalence class. In the case of assemblies there are two equivalence classes; resource equivalence and resource subsumption

(as explained previously in the Concept: Unary Operations). However as a result of such an operation, two criteria for assembly validity can be breached :

- **Acyclic condition:** Even though operand assemblies were valid, the quotient assembly containing merged resource dependencies can form cycles,
- **Conflict condition:** The union assembly containing resources and dependencies from the operand assemblies can have conflicting resources.

For the acyclic condition, the unary operations that contract equivalent or subsumed resources are already not closed on valid assemblies set $\mathbb{A}^*$, because of this condition. The problems caused by this property and their consequences are discussed later in this chapter. But for the sake of simplicity the cases join operation produces cyclic assemblies are omitted.

For the resource conflict condition, the join operation resolves such conflicts by replacing the conflicting resource in the right-hand operand with its counterpart in the left-hand operand. In a sense, as a result of the operation $A@B$ the assembly A is ***joined into*** the assembly B, and in case of resource conflict the resource description contained in A prevails over the resource description in B. In the context of the join operation, the second operand, assembly B is called the ***base assembly*** and the first operand, assembly A is called the ***joined assembly***.

The following procedure describes a possible implementation of join operation between two assemblies. Notice that before gluing two assemblies conflicting resources are registered and then they are contracted accordingly during the gluing phase.

Procedure join(assembly A, assembly B)

Input: $A = (V_A, E_A),\ B = (V_B, E_B) \in \mathbb{A}^*$.
Result: $R = (V', E') \in \mathbb{A}^*$ with $V' \subseteq V_A \cup V_B$ such that $\nexists r_1, r_2 \in V' \mid r_1 \bowtie r_2$.

```
1  forall the r1 ∈ V_A do // First register conflicting resource pairs
2      forall the r2 ∈ V_B do
3          if r1 ⋈ r2 then
4              C ← C ∪ (r1, r2)
5  R ← A ⊕ B // disjoint union of A and B
6  forall the (r1, r2) ∈ C do // Contract conflicting resources in R
7      R ← contraction(R, r1, r2)
8  R ← identify(R)
9  R ← subsume(R)
10 return R
```

The algorithmic description gives an idea on what the join operation does and how it can be implemented but it is not adequate for further analyzing the properties of the operation. Instead, it is then useful to describe the join operation using algebraic-categorical

approach. The operation can be defined as a graph morphism. Let A and B valid assemblies, there exists a graph morphism $\mathcal{T} : \mathbb{A}^* \to \mathbb{A}^*$, for ***transition***, that transforms B to $A@B$.

Furthermore for each graph morphism $\mathcal{T}$ that transforms B to $A@B$, a graph transformation system can be defined, guided by a set of production rules called T. The graph derivation $B \Longrightarrow A@B$ can be decomposed into a sequence of direct derivations $B = B_0 \Longrightarrow_{t_1} B_1 \Longrightarrow_{t_2} \cdots \Longrightarrow_{t_{n-1}} B_n = A@B$ such that the set of production rules T consists of $\{t_1, t_2, \cdots, t_{n-1}\}$.

Given that the join operation is applied on valid assemblies and supposing that the ***resource identification*** and ***subsumption*** operations are applied beforehand on both of the operand assemblies, the join operation can thus be reduced into the sequence of transformations that glue assembly A into assembly B, overriding conflicting resources. One can identify the ***types of production rules*** that are included in such a transformation system:

1. ***replace(r,r'):*** $p_{\{r \to r'\}} = (L \longleftarrow K \longrightarrow R)$. The production rule is described with $r \in L$ and $r' \in R$ such that $r' \equiv r$ or $r' \bowtie r$ or $r' \sqsubseteq r$. The graph R consists of r' and the edges (dependencies) of r' that are to be included in the target graph. The graph L consists of r and the edges of r that are needed to be in the original graph, to be able to apply the rule.

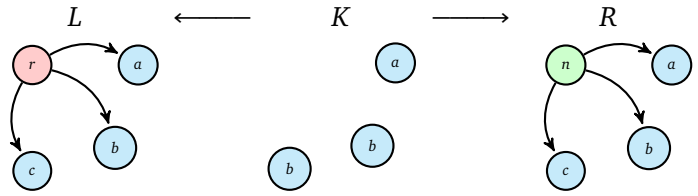


Figure 5.8: Example Production Rule for replace: $p_{\{r \to n\}}$

2. ***insert(r'):*** $p_{\{r'\}} = (L \longleftarrow K \longrightarrow R)$. The production rule is described with $r' \notin L$ and $r' \in R$. The graph R and L both include the edges of r, meaning that all the dependencies of r are already included in the original graph except r.

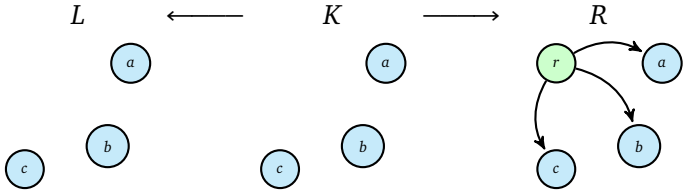


Figure 5.9: Example Production Rule for insert: $p_{\{r\}}$

Using these two types of production rules, a transformation system can be constructed such that each derivation $\Longrightarrow_p$ via the production rule p involves only one resource from the joined assembly. In each step of derivation, a resource of the joined assembly is ***joined*** into the base assembly. As a consequence, the set of production rules T and the order in which they are applied depends on the joined assembly i.e. the first operand, A in the example. This means that the transition function $\mathcal{T}$ can be generated from the joined assembly and thus it can be noted $\mathcal{T}_A$ such that $\mathcal{T}_A(B) = A@B$ (See equation 5.23).

$$A, B \in \mathbb{A}^* \, , \, \exists \, \mathcal{T}_A : \mathbb{A}^* \rightarrow \mathbb{A}^* \mid \mathcal{T}_A(B) = A@B \tag{5.23}$$

Sequential & Parallel Graph Transformation

In graph transformation systems an applicability condition describes if the production rule can be applied to a given graph. The applicability of a production rule ($L \longleftarrow K \longrightarrow R$) is defined by the left-hand side graph L. If the graph L is not contained in the original graph G, then the rule is not applicable onto graph G. According to the applicability rules, it is possible to speak of sequentially dependent and parallel independent production rules. Between parallel independent transformations the transformation system is locally confluent, meaning that independent from the order of application of those rules, the system will converge to the same result. Convergence and confluence of abstract rewriting systems are described in in [Church 1936] known as the Church-Rosser property.

In order to explain briefly, two production rules are sequentially dependent if one's applicability condition L involves vertices or edges created by the second rule. Similarly, they are parallel independent if their context graph K is disjoint (their applicability condition L and the inclusion graph R are disjoint).

Generating the production rule set is indeed straightforward, because for each resource in the joined assembly there will be a production rule.

The application order of these direct derivations is more complicated. Both of the above-mentioned production rule types allow generating rules that define non-empty left-hand side graphs. Naturally, the dependencies between generated production rules are same as the dependencies included in the joined assembly. For example, in the context of the operation $A@B$, a resource r contained in A, can be joined into B if and only if all of the resources on which r depends are already joined into B.

Given that the join operation is defined on the valid assembly set $\mathbb{A}^*$, the joined assembly is whether a directed acyclic graph $\mathcal{G}_a^d$ (DAG) or the empty assembly ε. In case of it is an empty assembly the transition function simply does no derivations. That is why

the ε is described as the identity element (see below). In case of a DAG, the order of the derivation sequence can be the strict partial order that is calculated by finding one of the transitive closures of the DAG according to dependency relations of the assembly. There are many well-known algorithms for finding partial orders of DAGs, especially used in scheduling, such as topological sorting algorithms. In the following example 5.10, assembly on the right is rearranged according to the distance of each node to the sink. An example derivation sequence would then be the $\{6, 4, 5, 3, 2, 1\}$.

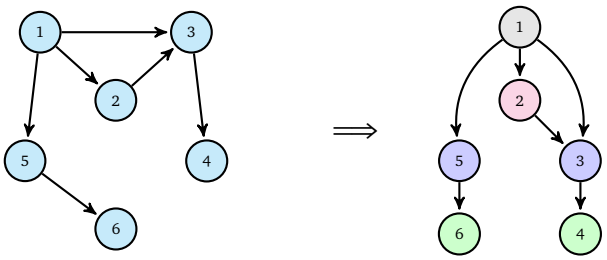


Figure 5.10: Calculation of Derivation Sequence

Note that there can be many different transitive closures that satisfy the dependency relation orders. Choosing one over another would not change the result of the operation. Depending on the dependency relations over the joined assembly graph, there can be derivations that are parallel independent, meaning that they can be applied in the same time, without any effect on the application of one another. In the previous example, using the parallel independence between resources of the same level, derivations can be regrouped as $[\{6, 4\}, \{5, 3\}, \{2\}, \{1\}]$.

A corollary of defining join operation as a sequence of transformations generated using the ***joined assembly*** is that at each application of direct derivation rule, the obtained result is still a valid assembly. This means that for every join operation $A@B$ via the transition function $\mathcal{T}_A$, there exists a transition function for each of the direct derivations such that its result is a valid assembly (See equation 5.24).

$$
\begin{aligned}
A, B \in \mathbb{A}^*,\ r_* \in R(A),\ \forall \mathcal{T}_A \mid\ & \mathcal{T}_A(B) = A@B, \\
& \mathcal{T}_A = \mathcal{T}_{r_n} \circ \mathcal{T}_{r_{n-1}} \circ \cdots \circ \mathcal{T}_{r_1} \circ \mathcal{T}_\varepsilon \\
& \mathcal{T}_A(B) = \mathcal{T}_{r_n}(\mathcal{T}_{r_{n-1}} \cdots \mathcal{T}_{r_1}(B))
\end{aligned}
\tag{5.24}
$$

Each one of these direct derivations $\mathcal{T}_{r_i}$ corresponds to the operation of the state transition function f defined resource types. With this algebraic definition of the assembly join operation in mind, the join procedure can be rewritten algorithmically as in the `join*`

operation.

Procedure join*(assembly A, assembly B)

Input: $A = (V_A, E_A), B = (V_B, E_B) \in \mathbb{A}^*$.
Result: $R = (V', E') \in \mathbb{A}^*$ with $V' \subseteq V_A \cup V_B$ such that $\nexists r_1, r_2 \in V' \mid r_1 \bowtie r_2$.

```
/* First identify and subsume A and B                    */
1  A ← identify(A)
2  A ← subsume(A)
3  B ← identify(B)
4  B ← subsume(B)
5  O ← partialorder(A)                  // Partial order of A
6  R ← B                                // Start from B
7  forall the r_1 ∈ O do
8      if r_1 ∈ V_B then
9          R ← replace(r, r', R)        // Apply replace rule
10     else
11         R ← insert(r, R)             // Apply insert rule
12 return R
```

Identity Element The identity element of the join monoid is the empty assembly, noted by ε. Joining an empty assembly with any other base assembly will result with the original base assembly as the derivation sequence will be empty (See equation 5.25). Similarly, if the base assembly is an empty assembly, the sequence of derivations will result with the joined assembly.

$$\forall A \in \mathbb{A}, \quad \varepsilon @ A = A \quad \text{and} \quad A @ \varepsilon = A \tag{5.25}$$

Associativity The assembly join operation is associative, meaning that the order in which the operations are applied does not matter, as long as the sequence of the assemblies is not changed. The definition of this associativity is shown in equation 5.26.

$$A, B, C \in \mathbb{A}^*, \ (A @ B) @ C = A @ (B @ C) \tag{5.26}$$

The proof of the associativity can be found in Appendix A. According to the general associativity theorem, the use of parentheses can be omitted for writing a given sequence of operands, such as $A @ B @ C @ \cdots @ D$.

Partial Commutativity The assembly join operation is not commutative, meaning that the sequence of operand assemblies changes the resulting assembly. This is reasonable considering the definition of the assembly join; because it overrides the

conflicting resources from the left operand. The definition of this commutativity is shown in equation 5.27.

$$A, B \in \mathbb{A}^* \, , \, A@B \neq B@A \tag{5.27}$$

In spite of that, in case of no conflict between A and B, the join operation is commutative. The proof of this is straightforward; without conflicts, the join operation is reduced to a gluing operation over identification and subsumption equivalence classes, which is commutative. This is shown in the following equation 5.28.

$$A, B \in \mathbb{A}^* \, , \, \nexists r_1 \in A \, , \, r_2 \in B \mid r_1 \bowtie r_2 \Rightarrow A@B = B@A \tag{5.28}$$

Together with the associativity, partial commutativity means that in case of there are not any conflicts between operand assemblies join operations can be applied in any sequence and order.

Idempotence The assembly join operation provides idempotence, meaning that any assembly joined with itself produces itself as the result (See equation 5.29).

$$\forall A \in \mathbb{A}^*, \, A@A = A \tag{5.29}$$

The idempotence property is obvious. The transition function generated by the assembly A would contain a series of transitions that for each resource they involve, there will be the same exact resource in A. The join operation will be reduced to a series of ***replace*** rules with same resources, which will leave the base assembly unchanged.

A corollary of the idempotence is that if the joined assembly is already ***contained*** in the base assembly, the operation would give the base assembly as the result. Another way of expressing this containment is that there exists two non-conflicting assemblies X and A such that the base assembly B is the join of these two assemblies. Than it is said that A or X is contained in B, and B will stay unchanged if it is joined with one of these assemblies (See equation 5.30).

$$\begin{aligned} A, B \in \mathbb{A}^* \, , \, \exists X \in \mathbb{A}^* \mid B = \mathcal{T}_X(A) \, , \\ \nexists r_1 \in A \, , \, r_2 \in X \mid r_1 \bowtie r_2 \Rightarrow A@B = B \end{aligned} \tag{5.30}$$

The assembly join operation presented here forms a monoid with idempotent, partially commutative and associative properties. These types of monoids are also called ***history monoids*** and they are often employed in computer science to model systems with a sequence of state changes. The sequential aspect of the join operation grant desired attributes to the deployment process and discussed later in this chapter.

Before proceeding to the next concept, a final point should be made about the acyclic condition of assembly validity. As stated before, a join operation between two assemblies

can create a cyclic assembly, even though neither one of the assemblies contained cycles. This is not an obstacle for the application of the join, because only the acyclic condition of the joined operation is needed for finding a sequence of transformations. Moreover there exist algorithms to break a cyclic graph into two or more acyclic graphs. Obtaining a cycle means that the resulting assembly cannot be joined later into another, which invalidates the associativity.

e. Platform

In the context of deployment, it is useful to define the concept of the deployment platform, also called the deployment site, on which the assemblies are deployed. A ***platform*** is a specialization of the assembly concept, containing resources and the dependency relationships. The special case for platforms is that they are valid assemblies and all of their resources are ***specified***. The platform is defined in the following equation 5.31.

$$P \mid P \in \mathbb{A}^* \quad \text{and} \quad \forall r \in resourcesOf(P), r \in \mathbb{S} \tag{5.31}$$

This broad definition of the platform concept allows to describe the deployment process in terms of assembly operations. The deployment process is, for the most part, a join operation between a platform and an assembly to be deployed. What sets the deployment apart from the join operation is that its result is a platform, meaning that all of its resources are specified. Yet a simple join operation glues two assemblies, contracting equivalent and conflicting resources but preserving any other type of resources such as inquiry and constructive. Instead, the deployment process includes an additional step to transform inquiry and constructive resources into specific resources before joining them into the platform. For the generated derivation sequence of each resource r contained in the joined assembly,

- if r is an inquiry resource, the deployment process **must find** in the platform at least one specific resource that subsumes r.
- if r is a constructive resource, the deployment process **must find** a specific resource that is equivalent to r, or it **must construct** the resource by applying necessary actions (such as install, configure, update, create) on the execution platform and eventually join the resource r', which subsumes r, into the platform assembly.

If any one of these actions cannot be done, the deployment process fails. The procedure implementing this process is given in the procedure `deploy`.

Procedure deploy(assembly A, platform P)

Input: $A = (V_A, E_A) \in \mathbb{A}^*, P = (V_P, E_P) \in \mathbb{P}$.
Result: $P' = (V', E') \in \mathbb{P}$

```
1  A ← identify(A)
2  A ← subsume(A)
3  O ← partialorder(A)
4  P' ← P
5  forall the r1 ∈ O do
6      if r1 ∈ 𝕀 then
7          if ∄r2 ∈ V' | r2 ⊑ r1 then
8              return fail
9      else if r1 ∈ ℂ then
10         if ∄r2 ∈ V' | r2 ⊑ r1 then
11             r' ← construct(r1)
12             P' ← insert(P', r')
13 return P'
```

The platform concept represents the current state of the deployment site. At the end of each deployment process the deployed assembly is joined into the platform. So that the resulting platform is the latest actualized state of the deployment site. More exactly, the platform stays always a ***complete assembly***, meaning that the dependencies are necessarily satisfied inside the platform.

The ***idempotence*** property is pivotal in the construction of the platform. The idempotent `join` operation makes it possible to know the exact state of the platform over the course of multiple deployment processes. In case a resource description cannot be constructed inside the platform, the deployment can be invalidated, knowing at which point the process is halted. The consequences engendered by this property are detailed below in the discussions.

5.2.3 Application Related Concepts

Concepts presented until here describe the algorithms that are used to deploy an assembly on a platform. These algorithms constitute the basis for the coordination of the deployment process. But the goal of this formalization framework is also to define a deployment process that is capable of adapting software with continuous deployments. The ***application*** and related concepts enable expressing variability on deployment and define the management of applications at runtime.

a. Condition

A condition is a predicate on a given inquiry resource whether it can be ***specified*** in a given assembly, most often in a platform. It consists of a *Boolean* value ⟨true,false⟩ and an inquiry resource r_i (See equation 5.32).

$$c = (b, r_i) \mid b = \langle \text{true}, \text{false} \rangle \, , \; r_i \in \mathbb{I} \tag{5.32}$$

Conditions are useful to validate an assembly, according to the state of resources contained within.

b. Repository

A repository is defined as a knowledge base that is capable of responding to queries composed of constructive resources, by returning deployable and/or configurable software artifacts, as depicted in Figure 5.11. In a sense it manages a one-to-one index of constructive resource descriptions and software artifacts. An example for this kind of repositories would be Maven artifact repositories, where coordinates of group id, artifact id and version are associated with artifacts.

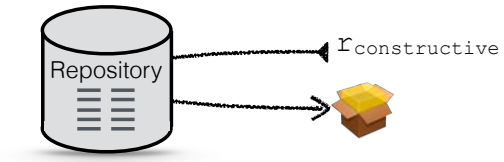


Figure 5.11: Repository

It is important to note that there are also repository implementations that incorporate dependency resolution capabilities. Such that, in terms of the concepts used in this conceptual framework, they are able to respond to queries composed of inquiry resources by returning an assembly, which contains all the dependencies transitively. OSGi Repository Admin, Eclipse p2 and YUM are examples of such repositories that are able to resolve dependencies.

The problem of dependency resolution is out of the context of this work; therefore this capability is deliberately excluded from the repository definition in this framework. Nonetheless such a mechanism can be used in tandem with the framework presented here as a means to complete assemblies that lack dependency information.

The artifact repositories are relevant in the context of application stores. A deployment system that allows to install software through an application store requires repositories for software artifacts and for application descriptions.

c. Application

The last concept of this formalization is the ***application***. An application description is defined to contain the information necessary for a deployment manager to deploy the application and also to manage its evolution later, during its execution. An application a is defined as a quadruple $a = (R, C_{Pre}, C_{Post}, A_C)$, composed of:

- R: A set of repositories. These repositories are to be queried by the deployment process to gather software artifacts to install or configure.
- C_{Pre}: A first set of conditions called ***pre-conditions***. These conditions should be valid for that the deployment of the application can proceed.
- C_{Post}: A second set of conditions called ***post-conditions***. These conditions should be valid at the end of the deployment and should continue to be valid along the lifetime of the application.
- A_C: A set of pairs of conditions and assembly (C, A) called ***conditional assemblies***. Conditional assemblies are deployed depending on whether the conditions associated with them are valid.

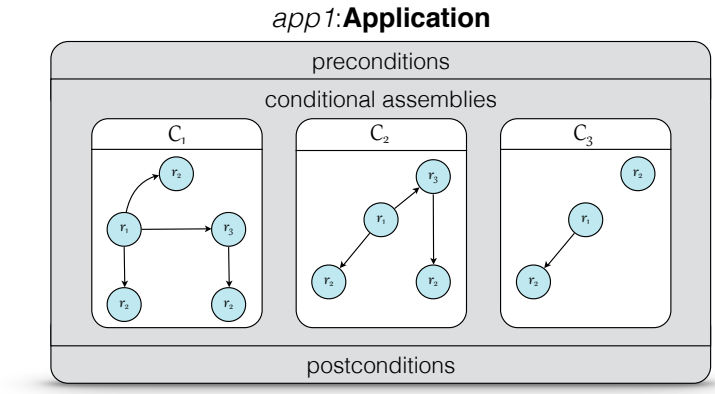


Figure 5.12: Application Example

In an application condition sets associated with each conditional assembly are neither disjoint nor exclusive. This implies that conditional assemblies of an application form a decision tree. At a given time, there can be several conditional assemblies for which all the conditions are valid. Deployment of an application requires obtaining a coherent assembly from all the conditional assemblies described by the application. This operation is called ***flattening***, and realized by joining together the assemblies with valid condition sets. Flattening operation is not possible if there are conflicting resources in different assemblies. Once a flat assembly is obtained, the deployment of the application proceeds by deploying the flattened assembly onto the platform. For an application deployment to take place there are a number of prerequisites:

- All pre-conditions associated with the application are valid.
- Post-conditions of the application do not contain a condition contradicting with post-conditions of existing applications.
- Flattening conditioned assemblies is possible and flattened assembly is valid and not empty.
- Existing pre-conditions does not contradict with the flattened assembly to be deployed.

The application concept augments the assembly concept in two ways. First and foremost, conditional assemblies contained in an application allow applications to change the way they are deployed according to the current state of the deployment site (i.e. actual state of the resources of the ***platform***). This adds variability to the application description and allows to reevaluate this description for further evolution of the application. In this respect the variability proposed by the application concept is very close to the ones studied in ***software product lines***.

Secondly, pre- and post- conditions serve to describe constraints on the deployment of the application. Pre-conditions designate constraints on the first deployment of the application onto a platform. As for the post-conditions, they become the invariants of the platform and validate the state of the resources of applications that are already deployed. Using post-conditions, for instance, it is possible to express that the state of a given resource will not be modified during execution. This guarantees that application deployments that occur afterwards will not alter the state of these resources.

5.3 Deployment Process

This section presents how all the concepts that are presented in the previous section is brought together in a process of application deployment. Lets consider the case of an application $a = (R, C_{Pre}, C_{Post}, A_C)$, being deployed on the platform $P = (V_P, E_P)$. And a set of existing post conditions C_P that are already associated with the platform P. Then the following steps are an example flow for the first deployment of the application a on the platform P.

1. The platform P represents the current state of the execution platform, i.e. the deployment site. The actual state of the execution platform is obtained either from a first provisioning of the deployment site or as a result of a sequence of deployments. The post-conditions C_P associated with the platform is the union of post-conditions of previously deployed applications and the constraints imposed by the platform operator in order to guarantee the integrity of the platform resources.

2. The application description a is composed of links to repositories, pre-conditions, post-conditions and conditional assemblies as expressed in concept 5.2.3.c. For a more controlled and secure deployment the developer company or some trusted partner maintains the linked repositories. These are to be used to gather the artifacts needed to construct resources. However, in another scenario the artifacts of the constructive resources can be bundled together with the description and transferred to the deployment site. Then one can imagine an earlier step that unbundles these artifacts along with the description.

3. The application description a is introduced into the platform P.

4. The first step of the deployment, is to check the eligibility of the platform towards the application. This is represented by the pre-conditions of the application. If all the pre-conditions return valid against the platform state, then the deployment can proceed.

5. Next the existing post-conditions C_P associated with the platform are checked against the post-conditions of the application. The deployment can proceed if no conflicts is found within these two sets of conditions.

6. In this step the deployment manager analyzes the conditional assemblies defined inside the application in order to decide which assemblies will be included for the deployment. This analysis produces a subset of conditional assemblies choosing the ones with all the conditions valid.

7. Next step is to produce a single coherent assembly from the set of assemblies chosen in the previous step. This operation, called flattening, checks first if there are conflicting resources in gathered assemblies and if there are not any, proceeds by

joining them together. For example, if the set of assemblies chosen as valid is $\{A, B, C, D\}$ then the flattened assembly would be $F = A@B@C@D$.

8. Before deploying the flattened assembly F the deployment checks whether it is valid. If the flattening did not reproduced a valid assembly, the deployment process return back to the step 6 and choose a new set of assemblies to be deployed.

9. Finally the deployment occurs as described in the platform deployment (see procedure deploy). The given set of repositories R are used to gather artifacts for the constructive resources. The new state of the platform is actualized as $P' = F@P$. The post-conditions of the application C_{Post} are also added to the post-conditions of the platform C_P.

Procedure deployApplication(Application a, Platform P, Set of conditions C_P)

Input: $a = (R, C_{Pre}, C_{Post}, A_C), P = (V_P, E_P) \in \mathbb{P}, C_P = \{c = (b, r_i)\}$.

```
1  forall the c ∈ C_Pre do
2      if check(P, c) then
3          continue
4      else
5          return fail                    // check preconditions
6  forall the c ∈ C_Post do
7      forall the p ∈ C_P do
8          if check(p, c) then
9              continue
10         else
11             return fail                // check postconditions
12 list[A] ← chooseAssemblies(A_C)        // choose assemblies to deploy
13 F ← ε
14 forall the B ∈ list[A] do
15     F ← F@B                            // flatten chosen assemblies
16 forall the r_1, r_2 ∈ V_F do
17     if (r_1 ⊑ r_2)||(r_1 ≡ r_2) then
18         F ← contraction(F, r_1, r_2)   // contract flat assembly
19 P ← deploy(F, P)                       // proceed to deploy
20 C_P ← C_P ∪ C_Post                     // update platform conditions
```

5.4 Discussions

Using concepts presented above, this formalization serves to outline a framework to define and implement the deployment process. It proposes concepts to represent the actual state of a deployment site, the expected state of what will be deployed and defines the deployment process that coordinates the actions to be taken to apply the expected new state. There are a number of choices that are made throughout the construction of this framework and of course these choices produce some desired characteristics and limitations. The following are the discussions on these choices, their consequences and limitations.

5.4.1 Actual vs. Observed State

Every deployment site, i.e. platform possibly contains a very large number of entities that constitute the actual state. In dynamic environments such as in pervasive computing, the states of entities are contingent and likely to change dynamically. Representing this state via platform and resources, like explained in this framework, requires continuous monitoring and reporting of the state of all the entities and building continuously the corresponding models that validate those. Constructing such models in computer memory can become very expensive and can easily disrupt the functional execution of the platforms business intent, notably deployed applications.

The deployment process relies on queries of whether the platform includes an entity that corresponds to some resource descriptions. Without a complete representation of the platform graph, the deployment process requires a framework that is capable of responding to such queries. This framework should be able to manage different types of resources and the state models associated with each resource. The implementation of the deployment process depends heavily on the capabilities of this framework representing resource states. The following section presents in detail this resource representation framework that is proposed for this purpose.

5.4.2 Idempotence & Determinism

It is previously argued that the assembly `join` operation is idempotent. For the deployment process this ensures that the coordination of actions that change the state of resources is idempotent. Still, in order to provide an idempotent deployment process each state transition f defined by resource types must also be idempotent. To recall, f is the transition function defined by each resource type to change the state of resources. State transition functions may involve multiple operations. For example, making sure that a software component is active may require first to transfer the executable binary, then install the component to the platform, and then configure it to active state. Resource type functions should at least guarantee that each state transition is ***sequence idempotent***.

This characteristic must be taken into account by the implementations of the function f provided by resource types.

Determinism can be guaranteed as soon as every algorithm used in the deployment process is deterministic i.e. returns the same value every time it is invoked with same input. This involves the resource type state transition functions, algorithms to analyze an application, algorithm to calculate the derivation sequence of deployment etc.

5.4.3 Traceability & Fault-tolerance

A process is traceable if every step and action taken in it can be identified and recorded chronologically. The fact that every state transition is well known in the deployment process makes it traceable. The sequential aspect of the join operation is crucial in this respect, because each transformation changes the state of the base assembly, by passing through valid, well known states. Regarding the deployment process, instead of applying state changes in a random order, the derivation sequence ensures that at each step a resource state described in the joined assembly is integrated as a specified resource in the platform. With this in place, platform operators can trace the evolution of the platform.

Determinism, idempotence and the traceability increase the fault-tolerance of the deployment process. Leveraging the traceability property, deployment actions can be coordinated inside a transaction-like construction where in case of an error; rollback actions can be applied in order to return resources to the previous state. Here, the term ***transaction*** is used in caution because ACID properties may not be satisfied in all instances. Contrary to the data-oriented systems like databases and filesystems, resources in dynamic execution environments (such as pervasive environments) are contingent, therefore not durable. Even though the deployment process ensures the atomicity (both the isolation atomicity and the failure atomicity) of state transition operations and their consistency; the state change can happen at anytime.

General idempotence of the deployment process also plays a role in providing a fault-tolerant deployment. Reapplying idempotent operations have no unwanted effects on the platform. So if the deployment fails due to an unhandled error, and the platform is in an unknown state, the deployment facility can restart the process. This approach can constitute the basis of distributed deployment system, in which idempotent deployment commands are coordinated for fault-tolerant deployment in distributed environments [Ramalingam 2013].

5.4.4 Reproducibility

A reproducible deployment process means that for a given target state, the deployment process can be applied in different starting conditions and will still produce the same state of the platform. Reproducing the same results (i.e. the same target state) with de-

ployments in different conditions is extremely important for deployment facilities. Every deployment site is different with unique disposition of resources. It is unfeasible to customize deployment requests for every other deployment site, especially when the scalability is at stake. In the continuous deployment paradigm, the same deployment description is deployed in multiple different target sites, like many testing, staging and production environments. The deployment process should reproduce the expected state described by assemblies, in spite of the heterogeneity of these environments.

The goal of achieving the target state should be evaluated in terms of the previous discussion on the "Actual vs. Observed state" (see 5.4.1). It is considered that a deployment is reproducible, or a deployment process is capable of reproducing its results, in the extent that it can produce platforms on different conditions but give same target state when observed. Other than that reproducing the actual state is unreasonable because each execution environment would contain uncontrollable parameters and characteristics.

There are two ways of ensuring that deployments are reproducible. First way is to calculate a proper set of operations on each of the different platform states so that the result of application of those will converge to the target state. Problem with this approach is to find the set of coherent, semantically and syntactically composable operations that will converge into the target state. It involves calculating all the possible configurations and distinguishes the paths that lead to the target state.

A second way is to make sure that each one of the different platforms undergo the same idempotent and deterministic state changes that represent the expected, target state. This second way is possible as long as the target state has a traceable path of state changes. The deployment process defined in this formalization achieves reproducibility using the second approach.

5.4.5 Application Compatibility

Every deployment facility should ensure two fundamental properties when it comes to the deployment of applications.

- ***Correctness:*** At each deployment request, the deployment process should make sure that the resulting state corresponds with the expected state of the application.
- ***Safeness:*** At the end of each deployment, the deployment process should make sure that the new deployment did not make any changes that invalidate or disrupt the correct execution of existing applications.

For correctness, the application concept defined in this framework ensures that a coherent configuration of the application (join of valid conditional assemblies) is deployed and that the post-conditions defined by the application are valid.

For the safeness of the existing applications, each deployment makes sure that none of the post-conditions will be invalidated at the end of the deployment process. Another way of putting this is that validity of the application against the future evolution of other resources on the platform can be guaranteed by post-conditions. For example, with post-conditions, an application can express that even though it includes a software component resource of a certain version, it will be still valid if that resource is updated, or downgraded inside a certain version range. This notion is often known as backward and forward compatibility. Likewise the application can also specify a post-condition to indicate that a particular resource should not be modified afterwards and keep the specified state. This kind of constraints are very common in platforms that contain core technical services that should not be modified by application deployments as in pervasive platforms or application servers.

5.4.6 Dependency Management

Dependency is a key concept in this formalization framework and there are important discussions about what kind of knowledge it can represent and how this knowledge can be obtained. During the presentation of the dependency concept it was already pointed out that dependencies only model the mandatory requirements of resources. But the semantic behind the notion of requirements can be various, depending on the types of source and target of the dependency. An example often appearing in other dependency systems is different dependencies software components are involved in. A software component may require services, executable binaries, software modules, but also may depend on a particular configuration of the container or execution environment on which they will execute. All these requirements of software components are indeed entities of execution environment and represented as resources in this formalization framework. Here the dependency relationship between the component and a particular container state represents a constraint on the execution of the component. Whereas dependency between two components represent a ***use*** relationship. It merits to be noted again that the deployment process does not actually wire links between dependent resources. This is delegated to the execution platform. All it does is to guarantee that the dependency will be resolved with a resource because it exists at a given required state.

The other important discussion about the dependency management, and the assemblies in general, is how does the knowledge about the dependencies is produced. During the development phase most of the executable software artifacts are created and packaged along with the metadata of their requirements at execution, such as execution constraints, required services and code libraries etc. Most of those dependencies can be extracted and resolved from those metadata. In some cases this information is not explicit and requires to be completed by human actors. For instance, a software component that needs a to read and write to a file, or to serial port requires that these resources are available for access.

Another case of the need for human intervention is when there are multiple solutions

for the target of a dependency. For the sake of determinism, the actor who specifies the deployment (usually developers or operators) should make decisions and create conditional assemblies that describe possible configurations of the application. As discussed previously in repository concept, resolving transitive dependencies for the target of a dependency is out of the context of this work. Most repository technologies incorporate mechanisms that resolve and return the transitive set of dependencies. Apache Maven, Eclipse p2, OSGi Repository Admin Specification and YUM are examples of such technologies. The deployment process assumes that all the dependencies are complete in the described assemblies.

5.4.7 Undeployment

Notice that there are no constructs or algorithms defined in the formalization framework to undeploy an assembly once it is joined into a platform. The terms ***undeployment*** or ***application uninstallation*** loose their conventional meaning because the deployment process is defined as a change of state of resources of a platform. Hence these terms need redefining. In terms of this formalization the undeployment of an assembly A can be defined as the join of another assembly $\neg A$ that undoes the changes made by the assembly A. Then the open question is how to calculate this $\neg A$ given that the platform could have undergone different deployments between the deployment of A and $\neg A$.

A naive approach to create this $\neg A$ would be to include the negative resources of only the constructive resources of A. It seems like this would uninstall all the resources brought (constructed) by A. This is fine as long as the assembly A's constructive resources are disjoint from all other resources of the platform, including other already installed applications. However, if the deployment of A reconfigures or assumes some already existing resource, as a result of a constructive resource it contains, then this approach is no longer reliable. The deployment of such $\neg A$ would compromise the safeness of the platform. An example to this case is the resources shared among different applications. Take the case of a platform P with two valid assemblies A and B representing two applications. If they do not contain any conflicting resources in between and with the platform, then A and B can be deployed with any order (conforming to the partial commutativity property described above). So the naive approach of undeployment would work without compromising the safeness of the platform. This case can be interpreted as if applications are ***isolated***, sharing only the platform resources via inquiry resources.

Even though it is out of the context of this work, resource sharing and application isolation are mechanisms that can be built on top of this conceptual framework. Sharing resources between applications, all by guaranteeing a safe undeployment process, would require extending the resource concept with properties expressing sharing policies. These policies then can be interpreted by a deployment process which will deploy applications in isolation, but also share resources between them as indicated.

Current industrial approaches of undeployment do not propose satisfying solutions to this problem. An early approach is to use defensive undeployment scripts. These scripts would uninstall applications, but leave any artifact that is likely to be shared. This is equivalent to writing assemblies by hand that uninstalls and deletes certain resources within a certain logic to undeploy one or several applications. In Apple Mac OS, application files are bundled into special packages. In theory the user can uninstall the application only by deleting this package file. However, in addition to execution processes, most of the running applications create files in different places of the filesystem for extensions, preferences, cache files etc. These files stay in the filesystem even though the application is uninstalled. A similar problem occurs in Microsoft Windows OS with registry entries and files. There are third-party applications that try to resolve this issue by finding and deleting related files.

5.4.8 Continuous Adaptation

The primary purpose of the application concept is to represent the notion of application at runtime. An application contains the necessary information to perform the first deployment and afterwards manage the application configuration during execution. The constraints on how the first deployment should be carried out is already discussed in application compatibility, section 5.4.5. As for the management of the application at runtime, it involves activities of deployment (creating new resources or changing the state of existing ones) and should be handled by the deployment facility.

In addition to the standard deployment process which alters the state of resources on the platform, runtime management of applications requires three important capabilities [Dearle 2007]. First is the ability to describe the resource configurations in which the application is still valid, or considered operating correctly. Instead of defining the application configuration as a static resource disposition, the application descriptor describes the variability of the application configuration depending on the conditions of the platform. This formalization allows two levels of variability:

- On assembly level, inquiry resources express the external dependencies. An inquiry resource can be described without precision, in order to accept many possibilities. For example an inquiry resource of type package can define a version range to accept packages of several versions.

- On application level, conditional assemblies allow to express different configurations of the application, depending on conditions, i.e. the current state of the platform. For example, a particular application feature can be installed or activated only if the platform is capable of executing it, or only if the user payed for such feature.

Second important capability is the ability to monitor the state of deployed applications. The application concept of this formalization framework does not define explicitly

a lifecycle, neither a state for applications. Given that each application is described with its own variability, it is not possible to define a common lifecycle and state for all possible applications. Then observing application state becomes synonymous with observing resources on which the application executes. Adequate monitoring policies are needed to decide which resources are to be monitored. A straight-out policy would be to monitor all the resources that the application description includes. However, not only this is not optimal, but also states of some of the resources are expected to change during execution. A monitoring policy advocated in this thesis is to monitor two aspects. First, the application post-conditions should be monitored for changes, because these are the invariants of the application and as soon as a post-condition is no longer valid, the application is invalid. Secondly, the changes in the resources involves in choosing the variants in conditional assemblies should be monitored to decide whether a conditional assembly is no longer valid or a new one become eligible. The implementation of resource monitoring is also problematic. Depending on the resource type, some resources may be capable of notifying on state change, while others need periodic checks whether their state have changed or not.

The third capability required for runtime management is the reevaluation of the application description. The application description with variability allows many different configurations for the application. A decision policy is needed to reevaluate different possibilities the variable application description proposes. This policy decides on a particular configuration and applies necessary deployment actions to change the application configuration. Once the monitoring detected a change, the application description containing variability should be reanalyzed, in order to choose another configuration. Here also there can be many policies that choose to deploy or not conditional assemblies. As expressed before, condition sets on conditional assemblies of an application form a decision tree.

This is very close to the autonomic control loop. Indeed the reference architecture presented in the next section is inspired by the MAPE-K architecture of autonomic computing [Kephart 2003].

5.5 Reference Architecture

This section presents the proposal for the reference architecture to implement the deployment process introduced in the section 5.3, using the concepts and operations presented beforehand in the formalization. The proposed reference architecture comprises two separate but complementary parts. The first is the context representation framework that serves to provide the current state of the execution platform. This module sits on top of the execution platform. The second is the deployment manager, which implements the described deployment process by using the representation provided by the underlying framework. Design details of these two entities and their primary functions are presented in their respective sections.

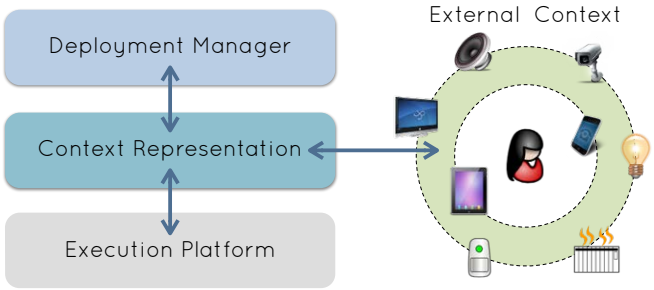


Figure 5.13: Layers of the Reference Architecture

Before presenting details of the architecture it is crucial to characterize the execution platform on which the reference architecture is based. The requirements on deployment platforms for implementing continuous deployment facilities are already presented in the previous chapter 4, as well as in the paper [Escoffier 2013b]. The ***modularity*** and ***configurability*** of the execution platform stands as fundamental properties required to implement this reference architecture. Then the continuous adaptation aspect of applications requires an ***introspectable*** platform monitoring, and the ability to make state changes by ***dynamically*** loading and unloading modules and ***reconfiguration on architecture-level***. The ***context representation*** however is considered inside the reference architecture. It provides the necessary information, such as the current state of the platform resources for the deployment process.

5.5.1 Context Representation

Notions of context and context-awareness are introduced in the chapter 2. To recall, a general definition of the notion of context is "*any kind of information that is relevant to characterize the situation of an entity*" [Dey 2001]. In dynamic environments, entities can be regrouped into three types of context entities, identified as computing environment, user environment and physical environment [Coutaz 2005]. The foremost challenge for reaching context-awareness in those environments is the construction of a representation

of the context and propagating changes to interested parties.

The context representation framework aims to provide an easy to use and uniform model for context sources to represent any type of context entity. Conforming to the deployment process concepts, these entities are represented and mapped as ***resources***. Resource's state, relationships with others, as well as actions to manipulate them are included in the resource representation. It allows context consumers to uniformly access the context, through resource representations, without any prior knowledge about the type of manipulated entities. To cope with dynamically changing context entities, it adopts the REST architectural style [Fielding 2000]. This section discusses advantages of this choice, followed by the details of the context model and the representation framework.

a. REST Architectural Style

REST is a software architecture style for distributed hypermedia systems such as the World Wide Web. It is based on a number of constraints for reflecting the properties of modern web applications such as scalability, fault-tolerance, recoverability, security and loose-coupling. While not undermining utility of these principles, the main interest is the uniform interface principle proposed by REST. According to this constraint, clients access resource representations through a simplified, uniform interface. However, this oversimplification may not suit to any application needs. In order to properly implement uniform interfaces, there are some constraints on the overall model:

- ***Resource identification***: A particular resource can be referenced by an identifier, regardless of its type or location.
- ***Resource manipulation***: Resource representations allow to retrieve the state of the represented entity as well as manipulating it.
- ***Self-descriptive messages***: Resource representations are self-descriptive; meaning they contain not only information about the resource, but also metadata that describes how the representation can be manipulated.
- ***HATEOAS*** (Hypermedia As The Engine Of Application State): Corollary to the previous constraint, client applications can examine resource representations that contain metadata about the state transitions and choose from alternative possible states, without prior knowledge about the type nor the structure of the resources.
- ***CRUD***: REST architecture style relies on the transfer of resource states. Even though it is not specified in [Fielding 2000], it is easier to guarantee state transfer and the interface uniformity, with a restricted set of operations. CRUD (Create, Read, Update, Delete) operations are chosen for a simple and universal way to allow both retrieval and alteration of those states.

Adopting REST provides a number of advantages for addressing the discussed requirements. Universal access and usability are greatly improved by the uniform interface. Self-descriptive resource representations enable providing a description of possible actions to manipulate the resource. Context providers can serve multiple versions of the context model, evolving and extending the context model without breaking existing consumers. Aside from these advantages, RESTful interfaces (CRUD operations or *Put*, *Post*, *Get*, *Delete* operations) are used commonly in development of Web APIs. This will create a positive incentive for developers towards implementing context representations.

b. Resource Model

The context representations are based on a resource model that represents dynamic context entities. A resource can be any context data. It serves as a representation of the state of the entity at a moment in time. The properties of the state are contained in the ***metadata*** of the resource, as key/value pairs. Resources are identified through their ***Path***. The hierarchical nature of paths creates a hierarchical composition of resources, meaning that every resource has a parent and may have ***subresources*** (children) logically attached to it. Therefore resources are organized into a tree structure starting from the root, all the way down to the leafs (Figure 5.14).

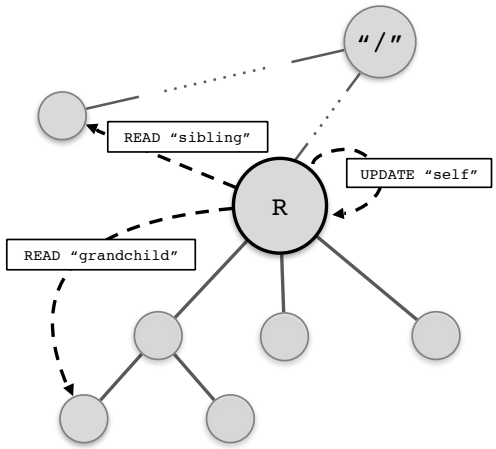


Figure 5.14: Resource Graph

The most important aspect for implementing a HATEOAS is the inclusion of links between resources that describe the state transitions. Just as web pages that contain hyperlinks, resources have ***relations*** describing links to other resources or themselves. Thus utility of relations is twofold: they describe how a resource should be manipulated, indicating operation type and parameters expected by the resource representation to apply this action. Also, they serve to link other resources, which constitute a virtual directed graph, where vertices are resources and edges are relations. It is possible to traverse this graph by retrieving the resources referenced by relations.

c. Resource Resolution & Observation

The main purpose of offering a context representation is to allow retrieving and modifying context state by applications. As depicted by figure 5.15, applications have two ways to interact with the context representation framework: ***requests*** and ***events***.

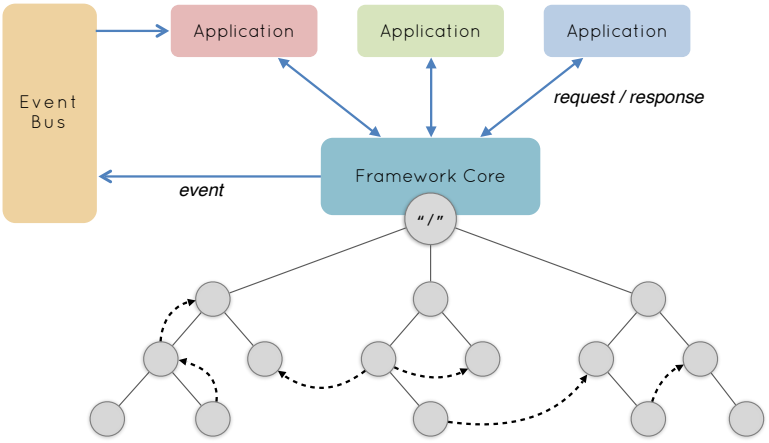


Figure 5.15: EveREST Framework Overview

Requests are sent by applications in order to retrieve the state of the context, or to impact it. Each request contains the intention of the requester to interact (using one of the CRUD operations) with a specific resource, identified by its path. The framework is in charge of the resolution of the resource, based on the provided path, and the application of the requested operation. Such operations may retrieve/alter the state of the targeted resources: their metadata and their relations. Therefore, requests offer applications a simple and unified way to interact with the context. Navigability through the context graph, using resource tree's natural hierarchy or customizable relations, is favored by the simplicity of the resource model.

While requests are a powerful way to interface with the context, they do not permit to capture all its dynamics. To achieve this crucial need, the framework augments the REST architectural style by emitting events on context changes. The framework can therefore notify applications that are interested in these changes, when such changes occur. Events are sent each time a resource is ***created***, ***updated*** or ***deleted***. Requests and events allow to impact on resource's state and to observe its changes, offering a fully dynamic representation of the context.

To improve the usability of the framework, requests and events can also use queries to select a set of resources. This feature gives the ability to mine inside the whole context to retrieve the adequate set of data. In addition, an application can be notified when a resource starts or stops matching a query.

d. Resource Extension & Transformation

The inherent simplicity of the resource model favors its usability. But much simplicity produces disadvantages, limiting the context evolution and flexibility. The framework provides two mechanisms to overstep these limitations: ***extension*** and ***resource transformation***. Addressing extensibility, it offers applications the ability to enrich the context representation by adding new resource spaces. By this way, applications can contribute to the context by providing new resources, accessible by other applications.

In addition to providing resources, applications can transform the metadata and the relations of resources to meet their own model. Such transformation can be contributed to the context, and kept private. The transformations are applied when a resource is referenced by a request of sent by an event.

Moreover, even though resources are untyped, which favors discoverability, it is sometimes useful to ***adapt*** the representation to the represented entity. Some resources give access to the underlying entity. Obviously, not all resources have this capability, but such feature promotes the connectivity between the context representation and its underlying objects.

5.5.2 Deployment Manager

The deployment manager implements the deployment process described formally in previous sections. It unites two main functions of the deployment process, namely, the analysis of the application descriptions and the coordination of the deployment actions. These two concerns of the deployment process are handled separately inside the deployment manager architecture, in ***Analyzer*** (section 5.5.2.b) and ***Planner*** (section 5.5.2.c) modules. To be able to provide these functions in a generic manner, the deployment manager also separates the modules that know how to execute deployment actions and how to observe and compare resource states. ***Resource Processors*** (section 5.5.2.a) are the extensions of the deployment manager that implement these resource interactions. The ***monitors*** and ***executors*** are created by the resource processors to interact with the context entities.

The ***context representation framework***, presented in the previous section 5.5.1, sits between the actual execution platform and the deployment manager. It provides a resource-based, unified interface to manage the entities of the deployment site. The Figure 5.16 presents the architecture of the deployment manager, alongside the context representation framework.

It is important to emphasize that the reference architecture adopts the MAPE-K architecture commonly used in autonomic computing and self-adaptive software systems. In terms of this architecture, the context representation framework constitutes the knowledge base. It provides the necessary information, such as the current state of the platform resources for the deployment process. Whereas the MAPE control loop, implemented by the deployment manager, ensures the deployment process and the runtime manage-

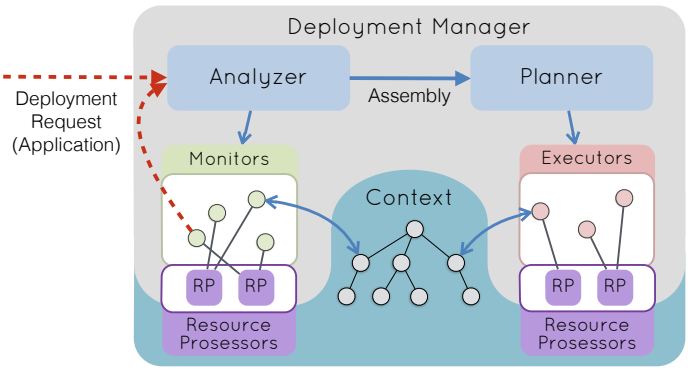


Figure 5.16: Deployment Manager

ment of applications. The figure depicts how the deployment manager's architecture uses MAPE-K as model architecture. This section continues by presenting the functionalities performed by different modules of the deployment manager.

a. Resource Processors

Resource processors are extensions to the deployment manager that are capable of manipulating resource states. Each resource processor is associated with a resource type. It is in charge of querying the context representation for resources and changing the state of resources of that type. Diffrent modules of a resource processor is shown in the figure 5.17. The primary function implemented by the resource processor is the state transition function, f, which is introduced previously in formalization concept 5.2.1.b). Resource processors also implement other type-specific functions needed during the deployment process. They provide two types of stateful components that are created and used during the deployment process and afterwards for the monitoring. These are deployment participants and resource monitors. Lastly, they provide extensions for the deployment descriptor language.

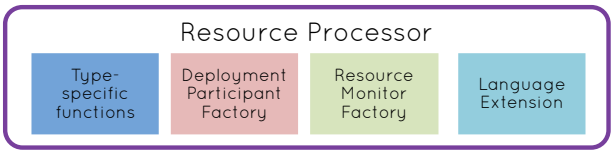


Figure 5.17: Resource Processor

Type-specific Functions Resource processors implement any function needed during the deployment process, the implementation of which is specific to each resource type. These functions include the resource subsumption, detection of resource conflict and fetching deployment artifacts and metadata of those, from given constructive properties.

Deployment Participants They are executor components that apply state transition

function f defined by the resource processor. Deployment participants are created and coordinated by the ***planner*** module. At each instance of deployment process, a number of deployment participants are created to apply unitary state transition on resources. Each deployment participant is therefore created with a target resource description and its state is only valid during the deployment process to which it belongs. It is responsible for ensuring that the given target state is attained by at least one specific resource. Their lifecycle is presented in the section 5.5.2.c.

Resource Monitors Resource monitors are components that observe and validate state of resources. The ***analyzer*** module creates resource monitors to validate particular application configurations and conditional assemblies. Each resource monitor is created with a condition. Depending on the type of the resource, it applies a monitoring policy on resources of context representation framework. They notify the analyzer if the given condition is no longer valid.

Language Extensions Language extensions are necessary for providing a deployment description language with different resource types. These extensions define valid properties and parsers for descriptors using processed resource type.

b. Analyzer Operations

The primary goal of the analyzer module is to treat deployment requests and calculate the assembly for deployment. Deployment requests can originate from the demand of new application deployment or for adapting existing applications. The analyzer module handles each application description separately to be able to apply the first deployment. Once deployed, the application is monitored and new deployment actions are calculated for adapting the application according to the dynamically changing context. The architecture of the analyzer is shown in the figure 5.18.

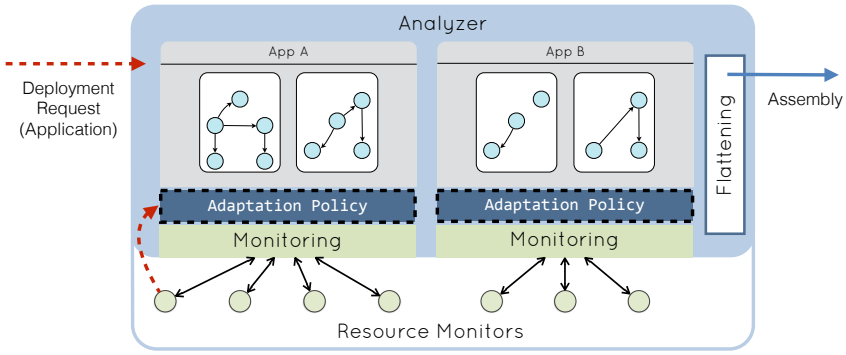


Figure 5.18: Analyzer Module

New Application Deployment Each deployment request contains one or more application descriptions. The deployment analyzer receives deployment requests and analyzes them to decide whether to start a deployment process or not. As described in

the deployment process, the analyzer first decides whether application description is eligible for deployment. This analysis includes checking pre-conditions of the application and choosing a subset of conditional assemblies defined inside application description. Here the analyzer is customizable with custom policies for each application. The ***adaptation policy*** is responsible for choosing a subset of conditional assemblies defined inside the application definition to be deployed. Then a flat assembly is calculated using chosen conditional assemblies. If a valid flat assembly is possible, this assembly is transferred to the planner module for deployment.

Application Monitoring As soon as the deployment of the flat assembly is started, the analyzer activates resource monitors necessary for observing the state of the deployed application. The analyzer associates a number of resource monitors for each managed application. As discussed previously in discussion 5.4.8, the resources to be observed are, by default, the resources that are involved in conditions of conditional assemblies and the post-conditions of the application. The resource monitors notify back the analyzer if a condition changes state from valid to invalid or vice versa. Depending on the type of the resource to observe, the resource monitor whether subscribes to the events of one or more resource entities, or checks periodically if the condition changed it's state.

Application Adaptation In case of condition changes, the analyzer handles the notification from the resource monitor and starts the adaptation process. This analysis comprises the reevaluation of conditional assemblies in the application description depending on the current, latest state of the platform. It calculates an assembly to deploy, which adapts the application to the current conditions of the platform, i.e. the context.

The main goal of application adaptation is to decide on a different application configuration, in terms of conditional assemblies that are ***effective***, i.e. deployed on the platform. There are two important aspects to consider in the implementation of the adaptation functionality. The first aspect is the set of policies that the analyzer will use to decide which conditional assemblies to include in for deployment. Self-adaptive systems and autonomic computing community studies self-star policies for optimizing, reconfiguring, repairing applications [Miller]. Analyzer enables implementing these policies inside the customizable ***adaptation policy*** associated with each application. In some cases, the adaptation policy can be unable to choose any valid assembly to apply onto the platform. This can either mean the application is unable to function in the current state of the execution platform, or it has fallen into an unrepairable state.

The second important aspect is the way the deployment manager transmits the application state from one configuration to another. To illustrate this, lets consider an application a that has four conditional assemblies, $\{A, B, C, D\}$, which is already deployed with the configuration $F = A@B@C$. And as the result of the reevaluation,

the adaptation policy decides on a new configuration that is $F' = A@D@C$. The deployment process should apply the state transition $F \Longrightarrow F'$ so that the assembly B should be ***removed*** from the platform and replaced by the assembly D. Then the analyzer should calculate an assembly G to deploy on top of F, such that $F' = G@F$. Indeed, the main problem of obtaining the assembly G is not with including D but removing B from the platform. This is partially an operation of undeployment.

Undeployment The problem of undeployment is already discussed previously in the discussions (see discussion 5.4.7). A solution is proposed in this reference architecture for undeployment of applications. Continuing from the previous example, undeployment involves constructing the assembly G as $G = F'@(-F)$. The ***negative function*** $(-)$ being a function that takes only the constructive resources contained in an assembly, and makes each of those resources ***negative*** by adding $\neg$ property. Then joining the negative of F with F' serves to override any resources shared in both F and F' along with the dependency relations described in F'. This way the resources of F' can be deployed using the correct dependency order, and the resources that were only constructed for the B can be undeployed. To achieve a complete undeployment of the application a, the same operation is applied with the $F' = \varepsilon$.

c. Planning Operations

The deployment planner module is in charge of executing the deployment command of an assembly. It receives assemblies to be deployed, enqueues them to the deployment queue and executes the deployment inside a transaction.

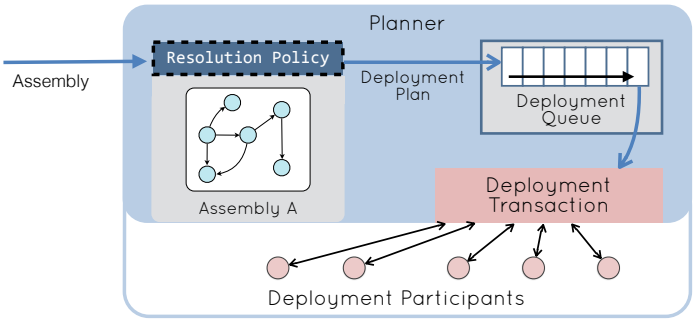


Figure 5.19: Planner Module

The execution of deployment starts by calculating the deployment plan, which is created in the basis of the transitive closure of the input assembly. Given that the input assembly is valid, meaning contains no cycles; it is always possible to find a plan. A deployment plan consists of a sequence of list of resource descriptions. At each stage of the sequence, state transitions of the resource list are parallel independent, so they can be executed in parallel. As discussed earlier in the assembly concept, there are different

algorithms to calculate this deployment plan. Here it is important for the sake of determinism to employ deterministic algorithms. A deterministic algorithm will result in exactly the same deployment plan, each time it is given the same assembly. The planner module allows to customize this algorithm, implemented by the ***resolver*** component.

Once the deployment plan is calculated for the assembly, the planner creates deployment participants with each resource description, using the resource processors respective to their resource types. Then the deployment participants are coordinated inside a deployment transaction according to the deployment plan.

Deployment Transaction Difficulties of implementing a transactional deployment are previously discussed in 5.4.3. Several assumptions are made in order to implement transactional coordination of the deployment inside the planner module. In an ideal transaction management system, transactions involve a sequence of atomic elementary actions. These actions are indivisible and either end successfully or fail. In the context of this work, instead of atomic actions, coordination elements are resource state transitions. So the first of the assumptions is that resource processors are implemented in a way that each state transition function f is a set of ***sequentially idempotent*** and ***fail-stop*** actions. This way the state transition is idempotent and its execution fails and stops the transition as soon as it encounters an error.

Other important point is the concurrency of transactions. Transaction management systems runs multiple transactions simultaneously by isolating their execution. Concurrency control of transactions requires serializing actions and managing their access to the resources via locking mechanism. The resource-based formalization allows applying these concepts to the deployment domain. Implementing locking mechanisms for entities in dynamic execution environments is difficult because of contingent nature of resources. Such mechanisms are outside the context of this work. Nevertheless, assuming that context representation framework provides such mechanisms; treating inquiry resources would need to acquire read accesses, while constructive resources would require write accesses. Without proper resource locking mechanisms, the second assumption is that there is only one deployment transaction executing on a platform. For this purpose the planner module employs a ***deployment queue***, which ensures treating one assembly deployment at a time (see figure 5.19).

Under these assumptions, the goal of deployment transactions is to extend the atomicity property over the deployment process. The atomicity property (in our case the ***failure atomicity***) allows limiting the uncertainty about the outcome of the execution of a transaction in the presence of failures [Krakowiak 2007]. A deployment transaction is thus defined for enabling recovery operations. It is modeled following the two-phase commit transaction protocol (see figure 5.20).

First the transaction coordinator calls all the deployment participants to ***prepare*** for transaction. At this phase participants decide whether the state transition is

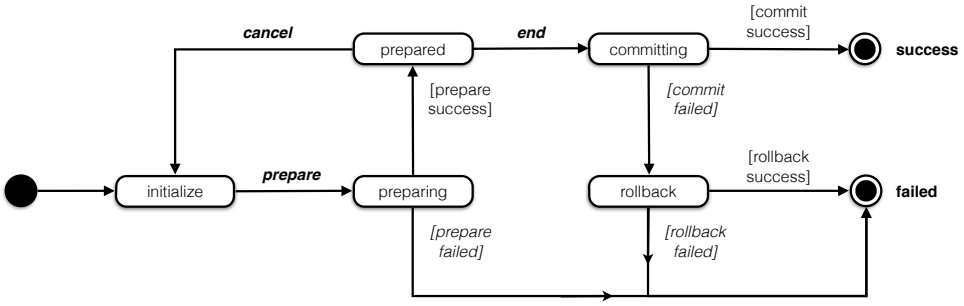


Figure 5.20: Deployment Transaction State Transition

possible (for constructive resources) and if it involves a specific resource, taking a backup the current state of the resource. If one of the participants cannot prepare the resource then the deployment transaction fails and all the participants that are prepared until that point are called to ***cleanup*** for aborting the transaction. If all participants prepared successfully then they are called to ***commit*** the state transition. Again if a state transition fails for one of the participants, or an inquiry returns with no results, the transaction fails. In this case the transaction enters the ***rollback*** phase, where all participants until the failed participant are called to rollback the resource to the previous state.

Error Handling Handling deployment failures is crucial for providing fault-tolerance. The deployment transaction ensures tracing the source of the fail and reporting it to the analyzer for further investigation. The transaction already includes the default behavior for handling deployment fails, which is the ***rollback*** phase. During rollback, the deployment participants ***try*** to recover from the modifications they made to the resources by undoing their actions. Therefore deployment participants need to save the initial resource state during the prepare phase. If all the participants rollback successfully, then the platform should be at its previous state. However, contrary to database management systems, in execution platforms rollback actions are not always possible, or they could also fail.

In both of these cases, the planner notifies the analyzer, the party who ordered the deployment of an assembly, with the state of the deployment, if it was ***successful*** or ***unsuccessful***, detailing the cause of the failure if there is one. In case of successful deployment, the default action of the analyzer is to activate monitoring components for observing the state of post-conditions. If the deployment was unsuccessful however, the analyzer can recalculate a new assembly, using the current state of the platform and the results of previous deployment attempts. This is an example of ***roll-forward*** and continuous software adaptation using information on historical deployment events. Such information facilitate developing autonomic policies that enhance applications with self-repair and self-optimization properties.

5.6 Description Language

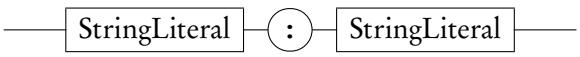
The last part of the contribution is the proposition of a domain-specific language (DSL) for describing the deployment of applications. This declarative language allows to express the concepts presented in the formalization until the description of applications. Application descriptions written in this language serve as an input for the deployment manager. Obviously on the basis of the proposed formalization, it possible to design different languages that describe the same concepts, using various constructs. Indeed, in the following chapter 6, in the context of this work two different implementations of this language is developed, each one with its advantages. However, they handle same kind of concepts. For illustration purposes the grammar syntax for one of these languages is presented here.

The deployment description language allows developers to codify the deployment process and treating deployment descriptors as first-class development artifacts. The description is stored in a file and archived on a version control system. It can then be analyzed for syntactic and semantic errors and transferred to the execution platform. This artifact is transferred to the deployment manager and treated as a ***deployment request***. This practice establishes the basis for the ***infrastructure-as-code*** movement and favors the continuous deployment [Spinellis 2012]. Leveraging the deployment process presented above, deployment events and results can be traced back to the deployment code developed in this language. Such development characteristics enable debugging and testing of deployment descriptors.

5.6.1 Basics

The description language is based on a number of syntactic atoms, which facilitate the definition of following constructs. First one of these elementary constructs is the ***property*** (Figure 5.21). It serves to define key, value properties as described in the property concept.

Property



Properties

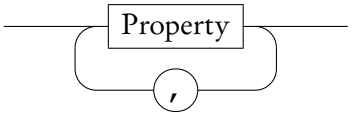


Figure 5.21: Property

For the sake of simplicity the concepts such as resources and assemblies in the formalization do not contain unique identificators. In the descriptor language, the ***Id*** construct

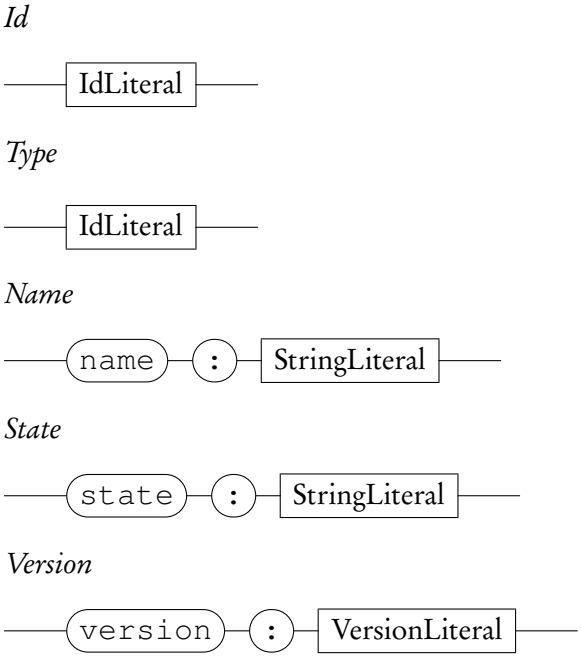


Figure 5.22: Id, Type, Name and State

is used to attribute identificators to some of the concepts (Figure 5.22). It is necessary to create references to language elements.

Similarly, the ***Type*** is also used to identify the resource types. Two common resource properties are also identified as ***Name*** and ***State***. The ***Version*** is also a common property required in deployment systems. There are many conventions about the versioning schemes. In this case the *VersionLiteral* can any expression that allows totally ordered versions.

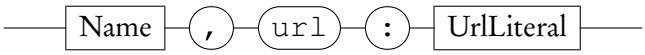
5.6.2 Repository

The ***repository*** construct contains the name and remote access URL for a repository, defined in the repository concept. The diagram in 5.23 shows the syntax of repositories and set of repositories. Again for the sake of simplicity, the security issues are left aside from this description. Thus the configurations needed for authentifications and secure repositories connections are omitted.

5.6.3 Resource & Assembly

As described in the resource concept, property names of resource descriptions are different for each resource type. The description language defines the syntax of resource description for a generic resource type, shown in the diagram in 5.24. The ***Name*** and

Repository



Repositories

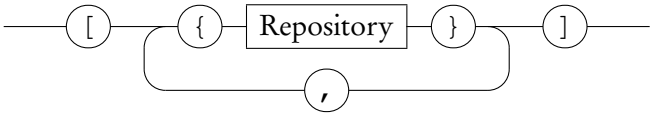


Figure 5.23: Syntax Diagram of Repository

State are common information for resource descriptions. To extend this language syntax, each resource processor defines the name of ***Type*** it handles and the names of its ***Properties***.

GenericResource

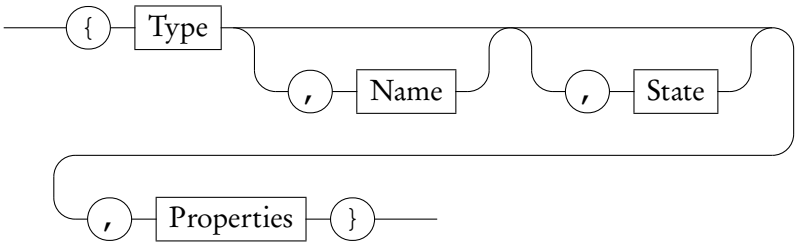


Figure 5.24: Syntax Diagram of Generic Resource

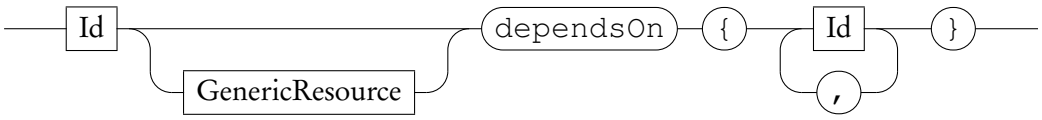
To form an assembly resource descriptions are declared and linked to each other by dependency relationships. The syntax of resource declarations and assemblies are shown in the diagram 5.25. Resource declarations attribute an identificator, ***Id***, to the resource descriptions. This identificator is unique for the resource declaration inside the assembly it is included.

Resource declarations comprise either a resource description or the identificator of an already included resource and refer to several other resource identificators as its dependencies. The ***dependsOn*** keyword is used to express the dependency relationship. Resource declarations are included inside an assembly by the ***resource*** keyword. This way inside an assembly description, dependencies can be either inline with resource state descriptions or they can be included all at once by referring to already described resources.

5.6.4 Condition & Conditional Assembly

As introduced in the formalization the condition concept, a condition is composed of a resource state description and a ***fact***, which is either true or false. The syntax of a condition contains a generic resource and a fact (Figure 5.26).

ResourceDeclaration



Assembly

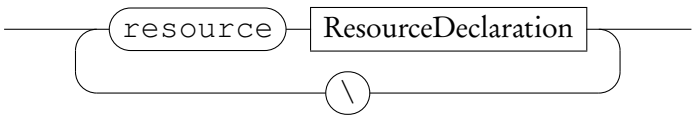
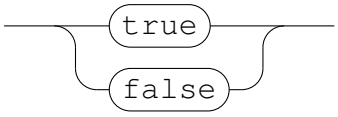
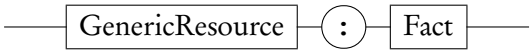


Figure 5.25: Syntax Diagram of Resource Declaration and Assembly

Fact



Condition



Conditions

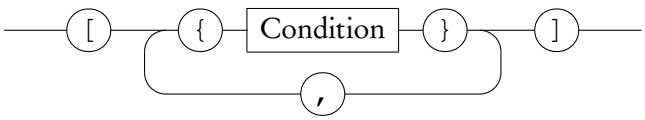


Figure 5.26: Syntax Diagram of Condition

Each application contains assemblies grouped inside a number of conditional assemblies, described in the application concept. A conditional assembly regroups a set of conditions and an assembly in order to include the assembly to the application. There is also the case when the condition set of a conditional assembly is empty. This means that the assembly is unconditionally added to the application. Therefore, the syntax for conditional assemblies considers these two cases. First the case ***with***, which includes the assembly to the application without conditions. And second the case ***when***, which includes the assembly described after ***then***, with the given non-empty condition set. The syntax describing these two options are shown in the diagram 5.27. Notice that this construct enable creating applications with variability.

ConditionalAssembly

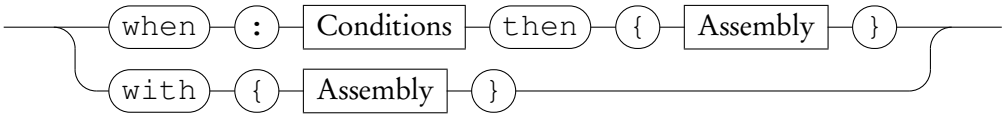


Figure 5.27: Syntax Diagram of Conditional Assembly

5.6.5 Application

Finally the parent construct for this descriptor language is the application. As mentioned earlier an application forms a deployment request for the deployment manager. A unique ***Id*** identifies each application description. Applications contain also a human readable ***Name*** and a ***Version***.

Depending on the policy of the deployment manager the (***Id***, ***Version***) pair can also be used as a unique identificator for the management of applications. Then it is up to the deployment manager to handle cases such as different applications descriptors with the same Id, multiple application descriptors with the same Id but different versions etc. These choices are deliberately excluded from the reference architecture, because they depend on the design decisions and the capabilities of the execution platform.

Conforming to the formalization in the (application concept), an application is composed of a set of repositories, pre- and post- conditions, and a set of conditional assemblies. The syntactic definition of an application is shown in the diagram 5.28.

Application

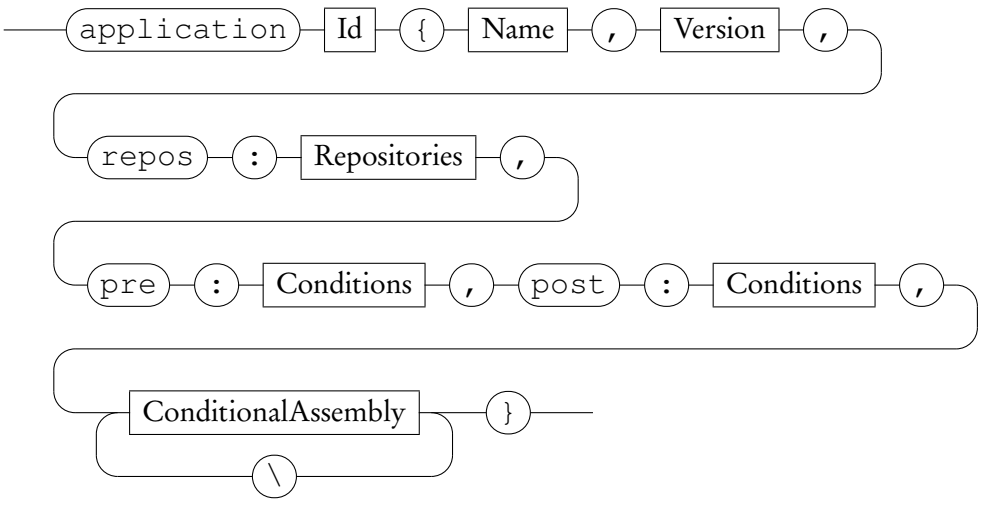


Figure 5.28: Syntax Diagram of Application

Ideally each descriptor file describes one application only. New deployment requests for an application (application update for example) can be described inside new descrip-

tor files referring to the application by its identificator. It is also possible using default language constructs to reference and include code from other files. This lets developers to write common assembly descriptions, stock them inside separate files and reuse those by including them to their application descriptor file. The grammar in EBNF language can be found in the Appendix B.

The following example describes a simple application with two included assemblies:

```
application "MyApp" , name: "My Application", version:"1.0.1",
    repos : [
        {name:"maven-central",
        url:"http://oss.sonatype.org/content/repositories/releases/"}
        ],
    pre: [] ,
    post: [] ,
    {
    when [
        {{file state:"exists", path:"/etc/bash_rc"}:true},
        {{bundle state:"ACTIVE",
            'bundle-symbolic-name':"org.apache.felix.eventadmin",
            version:"1.3.0" }:false}
        ]
    then {
        resource eventadmin { bundle
            'bundle-symbolic-name' "org.apache.felix.eventadmin"
            state "ACTIVE"
            'bundle-version' "1.3.2"
        }
    }
    with {
        resource eventadmin { bundle
            'bundle-symbolic-name' "org.apache.felix.eventadmin"
            state "ACTIVE"
            'bundle-version' "1.3.2"
        }
        resource log { bundle
            'bundle-symbolic-name' "org.apache.felix.log"
            source "mvn:org.apache.felix/org.apache.felix.log/1.0.1"
            state "ACTIVE"
            'bundle-version' "1.0.1"
        } dependsOn(eventadmin)

        resource logPackage { pkcg
            id "org.osgi.service.log"
            version "1.0.1"
        } dependsOn(log)
    }
}
```

Listing 5.1: Example of an application description

5.7 Evaluation

To conclude the proposition chapter, this section evaluates the presented contributions against other works in software deployment domain. In the first evaluation section, the formalization proposed in this work is compared to the previous studies that proposed formal frameworks. Secondly, major contributions of this work; the deployment process, the reference architecture and the description language are evaluated against the characterization framework, established in the chapter 4. And lastly this chapter concludes by summarizing the contributions of this work.

5.7.1 Comparison of formalisms

In this section a number of studies that propose formalization of deployment concepts are presented and compared to the proposition of this work. These studies are chosen for the evaluation particularly because they focus on solving different problems and thus for some employ different models for deployment concepts. The goal of this evaluation is to identify the similarities and differences of these approaches.

1. First in [Parrish 2001], authors lay out the general concepts for deployment of component based applications. They model ***components***, ***applications***, the current state of an execution platform (***configuration*** as called in the paper) and ***installations***, which is the deployment process for applications. Authors focus on managing different versions (or different implementations) of components and examine different strategies for replacing components. They identify these strategies as ***Replace Always***, ***Replace Only If Newer*** and ***Never Replace***. Then two installation properties are defined, as ***successful*** and ***safe***. A successful installation implies that the installed application works properly. A safe installation implies that existing applications continue to work after the installation is applied. The authors continue by defining backward and forward compatibility for components. And finally they link installation strategies with those in order to define the conditions for safe and successful installations. While this work defines many important concepts, it lacks the concept of dependency between components. There are no definitions or rules for the deployment process and the execution of applications.

2. In [Buckley 2005], Buckley provides a framework to resolve code dependencies and load those dynamically, specifically on the CLR (.Net) execution environment. The paper defines the ***assembly*** (a CLR core concept) structure that contains executable binaries and associated with metadata such as name and version and dependencies. The formal framework defines the deployment process of a module (an assembly) by resolving its dependencies, locating those and finally executing the module by making necessary bindings inside the execution context. The author details the

process of loading the assembly to the execution environment. Here each successful load operation changes the state of the execution context. During the loading of an assembly, if a dependency resolution fails to find a corresponding assembly, the install operation tries to gather it externally (from the end user), and tries to load that assembly on demand. If the system fails to make necessary bindings the installation fails, leaving the environment unchanged. This work concentrates on the internals of the runtime environment for resolving module dependencies by names and linking them. However, it does not address the evolution of applications. Similarly in [Escoffier 2006], authors explore dynamic code loading and unloading capabilities of .Net platforms.

3. Liu et al. propose a formal framework for modeling the deployment of component based applications [Liu 2006]. Their formal framework models the whole of the component deployment lifecycle, from *building* and *assembling* components through *shipping* the system from the development site, *installation* of the system at the deployment site, *reconfiguring* the system in response to changes and *executing* the system. The proposed formalism is based on the concept of ***application buildbox***, which is the resolution space for component dependency constraints. The labeled transition system they propose defines the state changes of this buildbox, therefore the evolution of the application. Their component definition let considering static and dynamic dependencies. At development time, components are packaged into ***assemblages***, which are modules with interfaces. Then the goal of the formalism is to ensure that the application builbox (thus the deployment site) is ***well-formed***, meaning that all the constraints on components and their dependencies are satisfied.

4. In [Belguidoum 2008] authors propose a formalization of component substitutability. Their goal is to provide a safe and flexible upgrade operation per component. The proposed formalization compares the dependencies of components as service interfaces and context descriptions. Components are described with mandatory, optional and negative dependency declarations. Then at runtime, wires between components satisfy these requirements. The context is defined as the current state of the execution environment. The work focuses on ensuring the safety of the system. To achieve this, the paper proposes verifying tthe requirements, the effect of the substitution and preserving invariants of services, components and context.

5. Lastly in his doctoral thesis, Sun examines the complexity of configuration management [Sun 2006]. The author proposes a state machine model for configuration management systems. On this model the reproducibility and composability of operations are studied. The proposition studies properties such as ***idempotence***, ***sequence idempotence***, ***commutativity*** and ***convergence*** of operations composed of atomic actions. Using this formalism, it is proven that in general cases system management processes are NP-complete and NP-hard. Then the process of depen-

dency analysis between managed entities and operations is described. Here the distinction between two types of entities is made. In the ***black-box*** approach, only information on external behavior is available. As for the ***white-box*** approach, entities contain some representation of content that is available for analysis. High complexity of these processes causes for the configuration management to be non-deterministic and intractable for system administrators.

Compared to those formalisms, the proposition in this work stands out in several aspects. To begin with, the formalism in this work specifically aims to coordinate the deployment operations. In this respect, the core concept is chosen as the ***resource*** to be able to cover different kinds of entities that can be found on deployment sites. Other formalisms except the 5 are limited with component-based deployments.

Continuing from the coordination aspect, the studies in items 2, 3, 4 specify the internal operations of the execution platform required for installing and executing components. However, this work proposes to delegate that concern to the underlying platform and only command state changes of resources. Specifically the details discussed in the work 3 for component installation can be used to implement the resource processor for components.

This work adopts the ***declarative approach*** for describing the deployment of multiple entities. So users provide the deployment descriptor that designates the expected state of one portion of the execution platform. The state transition process that makes this happen is then deduced from this description. As argued in 2, this is the ideal approach for verifying the state of the platform. Other works, while describing also components, concentrate primarily to resolve the deployment concerns on the basis of single components.

Contrary to the 3 and 4, where dependencies can have different types and contain constraints, this work, as like the one in 2, consider only ***simple dependency*** descriptions. This reduces the complexity of graph operations, because constraints on entities are only described in resources.

The formalisms in 2, 3 and 4 concentrate on resolving dependencies with ***constraint-solving and name matching***. The complexity of such algorithms that choose components by constraint-solving is studied in 5. Constraint satisfaction algorithms often use heuristics for limiting and reducing the resolution time and therefore not deterministic. In this work most of the dependency choices are made beforehand, when the deployment descriptor is created. Surely the application description contains different conditions, which are constraints on the platform state. But the resolution of those constraints and decisions on application variability are confined to the analyzer module. The current algorithms can be replaced by sophisticated constraint solvers such as SAT-based engine with or without backtracking.

Lastly, the ability to describe variability allows applications to ***adapt*** to the changes on the platform state, and ***evolve*** on the long-term. Continuous adaptation proposed in

this work eliminates the restriction of declarative approaches, which is also expressed in the work 4.

5.7.2 Evaluation for Continuous Deployment Requirements

Following the formalism comparison, this section evaluates the continuous deployment requirements satisfied by as the result of this work. To recall, in the previous chapter 4 requirements for continuous deployment are presented in three groups, deployment platform requirements, deployment process requirements and language requirements.

Platform Requirements As it is discussed previously in the section 5.5 that the proposed reference architecture presumes an underlying execution platform. This execution platform is supposed to be configurable, introspectable, modular, dynamic and capable of applying architectural reconfigurations. Therefore, the requirements defined by the characterization be present on the platform, except the context-representation. This last requirement is satisfied by the reference architecture by the proposition of context representation framework presented in section 5.5.1.

Process Requirements The notion of deployment request appears in the proposed architecture as the application description. Application descriptions trigger analysis and then the execution of deployment inside deployment manager. The deployment request can be introduced to the deployment manager (more specifically to the analyzer module) either from outside by installing or updating an application (***push***), or from inside the deployment manager, as a result of change demands of a already installed application (***pull***).

The proposed deployment process ***determinisim***, ***idempotence*** and ***fault-tolerance*** properties are already argued in corresponding discussions 5.4.2 and 5.4.3. These reflections are then transferred into the reference architecture for the proposition of idempotent resource processors, deterministic deployment plans and transactional process execution.

The proposed process and architecture are ***customizable*** in many points. The generic process allows to integrate new resource types, and this is also ratified in the reference architecture by the resource processors. Algorithms and policies with many possible implementations are left customizable inside the deployment manager architecture.

The reference architecture is proposed specifically to support the ***continuous adaptation*** of applications. Each deployed application is monitored automatically and changes are notified to custom application policy for adaptation decision. Changes decided by the adaptation policies also pass from the same coordination process as regular deployments.

Language Requirements The description language proposed in the section 5.6 allows to express resource descriptions according to the concepts defined in the formalization. As a result, the language allows to describe resource states and their dependencies. Leveraging the use of different description levels, the description language lets users define their applications with more or less precision.

The language is ***extensible*** with different resource types. Resource processors are in charge of providing implementations of language extensions for the resource type they manage.

Aligned with the application concept in the formalization, the description language allows describing conditional assemblies inside application descriptions. This enables ***variability*** over descriptions and is necessary for continuous adaptation.

Using standard language constructs, assembly descriptions can be referenced from different source codes. This eases the management of the deployment description codes and allows ***reuse*** of common portions of application descriptions.

The following table summarizes the evaluation of the proposition against these requirements. The ● signifies that these characteristics are inherited from already existing work. The ✔ represents the requirements to which this thesis have proposed contributions.

Table 5.2: Positioning against continuous deployment requirements

Platform		**Process**		**Language**	
Configurability	●	Pull/Push	✔	Expressivity	✔
Reflection	●	Determinism & Idempotence	✔	Extensibility	✔
Modular + Dynamic Execution	●	Fault-tolerance	✔	Variability	✔
Architectural Reconfiguration	●	Customizability	✔	Usability	✔
Context access	✔	Continuous Adaptation	✔		

5.7.3 Conclusion

This evaluation concludes the proposition of this thesis.

In order to summarize, the section 5.2 proposes a ***formalization framework*** for deployment concepts. This framework describes algorithms that coordinate deployment actions. The discussions in the section 5.4 explained the consequences of this deployment process. Concepts along with the deployment process are embodied inside the ***reference architecture***, presented in the section 5.5. The reference architecture describes the context representation framework and the deployment manager which implements the proposed process. Finally, a ***deployment description DSL*** is proposed in the section 5.6. This language allows to code deployment descriptions that serve as deployment requests for the deployment manager.

In the beginning of this chapter a number of objectives are presented to be addressed by this work. This evaluation explains how presented contributions satisfy their objectives. Following table 5.3 summarizes of the research objectives and corresponding contributions.

Table 5.3: Research objectives and contributions of the proposition

Objectives	**Contributions**
Reproducibility	Traceability 5.4.3, Determinism 5.4.2
Fault-tolerance	Transactions 5.5.2.c, Idempotence & Determinism 5.4.2
Continuous Adaptation	Application Description with Variability 5.2.3.c, Reference Architecture 5.5.2.b,
Tooling	Description Language 5.6

Chapter 6

Implementation and Usage

"I hear and I forget. I see and I remember. I do and I understand."

— Confucius

Contents

6.1 Introduction

The goal of this chapter is to present how the propositions of the previous chapter are implemented. Specifically, it presents the development projects of the context representation framework, ***EveREST***, and the deployment framework that is developed to validate the contributions of this thesis, ***Rondo***. The tools provided by Rondo includes the deployment manager that implements the reference architecture and the DSLs (domain-specific languages) for describing deployment of applications. Both of these projects, Rondo and EveREST, are developed on top of OSGi™ and Apache Felix iPOJO™ technologies. They are available as open source and are fully operational at their current state.

The following section presents the implementation details of the EveREST framework and different tools developed within Rondo tool suite.

6.2 Implementation

This section presents EveREST and Rondo frameworks, the global architecture they are used in, and later details each one of these projects. The EveREST project contains EveREST Core, EveREST OSGi, EveREST iPOJO, EveREST System and EveREST Filesystem. The Rondo project implements tools for the deployment, including the Rondo Core, Rondo Deployer, different Resolvers, Rondo Cloner and DSLs in Java and Groovyy languages.

6.2.1 Global Architecture

The main solution developed in the context of this work is the deployment framework called ***Rondo***. Rondo framework proposes a set of tools that implement the contributions presented during the previous chapter. These tools are developed for obtaining experimental results and for validating the approach of this work. The central tool of this approach is the deployment manager, implemented in the ***Rondo Deployer*** module. As proposed in the reference architecture, the deployment manager depends on a context representation framework for observing and manipulating the actual state of the platform. This context representation framework is developed in a separate development project called ***EveREST***. This section presents briefly the EveREST framework, followed by the details of different modules and tools proposed by Rondo framework.

Both of these projects are based on OSGi[1] and Apache Felix iPOJO[2] technologies. The OSGi is a modular service execution platform on Java technology. As discussed previously in 3.5.4, it is used as the basis of many deployment solutions. However, the evaluation of existing deployment solutions in chapter 4 concluded that the OSGi lacks the architectural reconfiguration support and a proper context representation. iPOJO addresses one

[1] http://www.osgi.org
[2] http://www.ipojo.org

of these issues, the architectural reconfiguration, by providing a service-oriented component model on top of OSGi service and module layer. The component model proposed by iPOJO manages the lifecycle of components, component instances, their configuration, execution and reconfiguration. It provides a simple programming model, hiding the complexity of the dynamism management [Escoffier 2013a]. iPOJO allows to configure instances with extensible mechanisms that select and inject service dependencies. These changes are applied transparently to the component code, at the architectural level. The following schema displays basics of iPOJO component model (see figure 6.1).

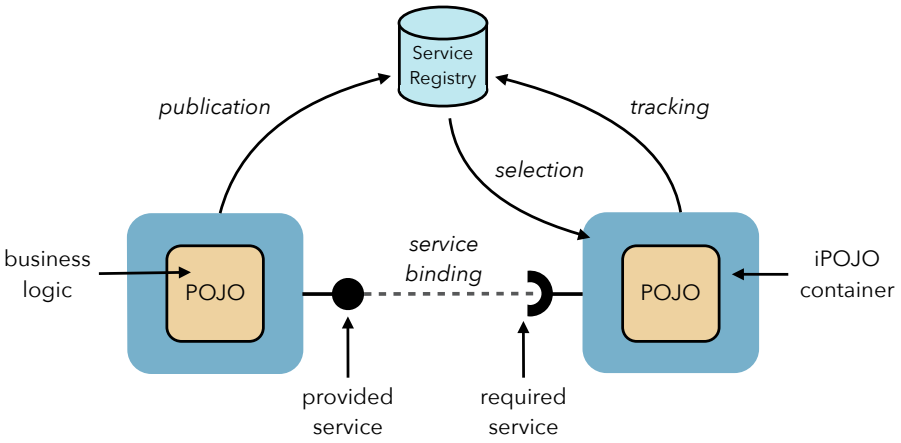


Figure 6.1: Apache Felix iPOJO component model

The other platform requirement lacking from OSGi is the context representation, which is addressed by the reference architecture proposed in the contributions and implemented by EveREST framework.

The deployment tools proposed by Rondo framework address primarily the applications running on OSGi and iPOJO. But as it is explained later in this chapter, the architecture of these tools allow to be extended with different types of resources and thus are not limited to this environment. This way Rondo and EveREST can be used in different domains without much effort. Following are the different sub-projects that are developed within Rondo framework. The diagram in figure 6.2 shows their interdependencies.

Briefly Rondo framework includes following projects:

- The ***rondo-core*** project provides the implementations of the resource and assembly models for the deployment descriptor. It proposes a fluent API for creating these models, which constitutes the Java DSL for the deployment descriptor.
- The deployment manager is implemented by the ***rondo-deployer*** project. This module provides all the necessary components for handling new deployment request programmed in DSLs and, for handling the deployment and management of applications.

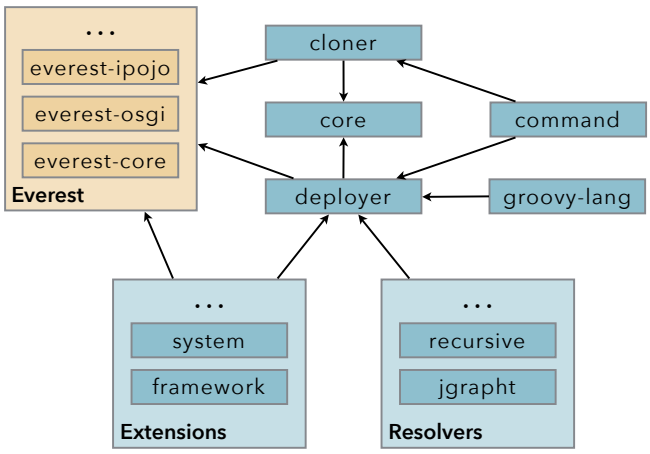


Figure 6.2: Project Dependency Graph

- The ***rondo-cloner*** module proposes a reverse-engineering tool for extracting the current state of a platform into a deployment descriptor. This descriptor can than be used as a basis for reproducing the same conditions of the platform.
- Certain resource types that are already built-in the core and the deployer modules allowing for their description and deployment. ***Extensions*** augment the description language and the deployment manager with other resource types.
- As expressed earlier in the proposition, there are many ways to calculate a deployment plan from a given deployment descriptor. ***Resolvers*** provide different algorithms for this important operation inside the deployment process.
- The ***rondo-command*** module proposes useful commands to the user for introspecting managed applications and the state of deployments, starting new deployments or cloning the platform state. It is useful for debugging purposes, rather than a tool for production.
- The ***groovy-lang*** module provides the deployment descriptor DSL on the Groovy language support. Groovy enable compiling and loading the code at runtime. Also it provides a more cleaner language syntax.

This chapter explains some of these projects in more detail, starting from the EveREST framework.

6.2.2 EveREST

The EveREST framework implements the context representation required by the deployment manager. The reference architecture of this framework is already presented previously in the section 5.5.1. The EveREST framework is composed of a ***core*** module and

several ***domains*** that extend this core. Domains provide resource representations of different kinds of entities. In its current state, the EveREST project contains domains for representing the execution environment of the platform it runs on. Having runtime models of the execution is pivotal for the implementation of the deployment facilities. These domains are OSGi, iPOJO, Java Runtime Environment and the Filesystem.

a. EveREST Core

The EveREST core provides the resource model that allows to apply REST architectural style to context representations. It is the essential part of the framework that serves as a bridge between domains and external applications that want to access and manipulate the context. EveREST maintains a common access point for external applications to make requests to resources. The `EveRESTService` interface allows applications to transfer requests to resources. A `Request` is composed of an action, a target path and a set of parameters. EveREST guarantees the tree structure of resources for that requests can reach addressed resource representations. To enable this, each `Resource` implements a `process` method, which by default delegates the request until its destination where it is finally treated and applied.

Each resource is an object that allows to obtain and manipulate the state of the entity it represents. Resources describe their own capabilities via relations they possess. A `Relation` is described with a name, a target `Path`, an `Action` and a set of parameter descriptions. In addition to that, some of the resources can produce notifications that inform applications of their state changes. EveREST allows resources to publish synchronous and asynchronous notifications, which relies on OSGi Event Admin messaging backend.

Domains provide implementations of the `Resource` interface for different kinds of entities they model. Each domain must identify itself with a *root* resource, which identifies the entry point to the domain. EveREST provides default implementations of the resource model. By extending these default implementations, domain developers can inherit the default behavior and structure of resources. This way they can concentrate on the model of their resource representations. Each domain model must design carefully the information they need to include, its resource structure and the relationships that resources will have.

b. EveREST OSGi

The OSGi domain models entities found in standard OSGi platforms. The entities modeled as resources include the configurations of the *OSGi framework* itself, *bundles*, *packages*, *services*, Configuration Admin *configurations*, *log entries* and Deployment Admin *deployment packages*. Resource models also include the relationship between different types of resources. For example, a bundle resource is linked to package and service resources it provides and requires. The listing 6.1 shows a bundle representation in Json. Deployment packages are linked to the bundles it contains. In addition to representing the state

of these entities, some resources let manipulating the state of resources. OSGi domain let installing new bundles and deployment packages, changing the state of bundles and creating and updating configurations.

```
{
  "bundle-id": 1,
  "bundle-state": "ACTIVE",
  "bundle-symbolic-name": "org.apache.felix.configadmin",
  "bundle-version": {
    "major": 1,
    "minor": 8,
    "micro": 0,
    "qualifier": ""
  },
  "bundle-location": ".../org.apache.felix.configadmin-1.8.0.jar",
  "bundle-last-modified": 1403801594012,
  "bundle-fragment": false,
  "__observable": true,
  "__relations": {
  "Child:services": {
     "href": "http://localhost:8080/everest/osgi/bundles/1/services",
     "action": "READ",
     "name": "Child:services",
     "description": "Get the child  \"services\"",
     "parameters": []
  },
```

Listing 6.1: Resource representation of a bundle

c. EveREST iPOJO

The iPOJO domain models the entities found in the iPOJO component model such as *component factories*, *component instances*, *handlers* and *declarations*. iPOJO domain is an example of how an EveREST domain can extend another. Resources in iPOJO domain contain relations to the resources of OSGi domain. The component factories and handler reference the bundle they are defined inside. Component instances reference the OSGi services they require and provide. It is possible therefore traverse the complete resource graph following these cross-domain relations.

iPOJO domain allows to create, reconfiguring and destroying component instances. The following instance representation in listing 6.2 contains relations for reconfiguring and destroying the instance. The important difference compared to architectural reconfiguration mechanisms is that the changes in EveREST resources are expressed as new resource states and not actions. Each resource representing the entity decides the actions to perform based on the current state and requested target state.

```
{
  "name": "org.ow2.chameleon.everest.core.Everest-0",
  "factory.name": "org.ow2.chameleon.everest.core.Everest",
```

```
  "factory.version": null,
  "state": "valid",
  "configuration": {},
  "__observable": true,
  "__relations": {
    "reconfigure": {
    "href": ".../org.ow2.chameleon.everest.core.Everest-0",
    "action": "UPDATE",
    "name": "reconfigure",
    "description": "Reconfigure this component instance",
    "parameters": [
        {
          "name": "state",
          "type": "java.lang.String",
          "description": "The state of the component instance",
          "optional": true
        },
        {
          "name": "configuration",
          "type": "java.util.Map",
          "description": "The configuration of the component instance",
          "optional": true
        }
      ]
    },
    "delete": {
      "href": ".../org.ow2.chameleon.everest.core.Everest-0",
      "action": "DELETE",
      "name": "delete",
      "description": "Destroy this component instance",
      "parameters": []
    },
```

Listing 6.2: Resource representation of an iPOJO instance

d. EveREST System

The System domain represents the properties obtained from the operating system and the standard Management Beans offered by the JVM, according to the JMX specification [Sun Microsystems 2006b]. System and environment properties, information on the operating system, memory and processor load and Java threads are represented as resources inside the System domain. Most of these resources are for monitoring purposes so they are read-only.

e. EveREST FS

The filesystem domain represents the local files and directories accessible by the platform. It presents an example on how the resources representing underlying entities can be created on demand, according to the request. Searching and creating every file and directory

on the filesystem is inconceivably inefficient. Filesystem domain creates the resource and its tree hierarchy once it receives a request for a resource. The issue of loading big resource graphs into memory can appear in different domains. The optimization used by filesystem domain is an example to circumvent this issue.

Before continuing to the implementation of Rondo framework, it merits noticing that the resource models offered by EveREST domains remarkably help implementing context-aware applications. The deployment facility provided by Rondo is an example for context-aware application. Rondo deployment manager deploys applications according to the current state of platform, represented by EveREST. The generic `Resource` interface allows to manipulate entities through a uniform interface, without necessarily knowing how to change resource states of different types of entities. Applications accessing context information concentrate on the information they possess, not the actions they should apply in order to obtain and change the state of entities. Using cross-domain relationships, it is easier to extract the information hidden inside links between different types of entities. All these elements ease the implementation of deployment tools provided by Rondo, which is presented in following sections.

6.2.3 Rondo Core

Rondo Core project implements the model of concepts presented in the previous chapter. This model enables creating the deployment description that holds information about applications, assemblies and resources. The diagram in Figure 6.3 presents this model.

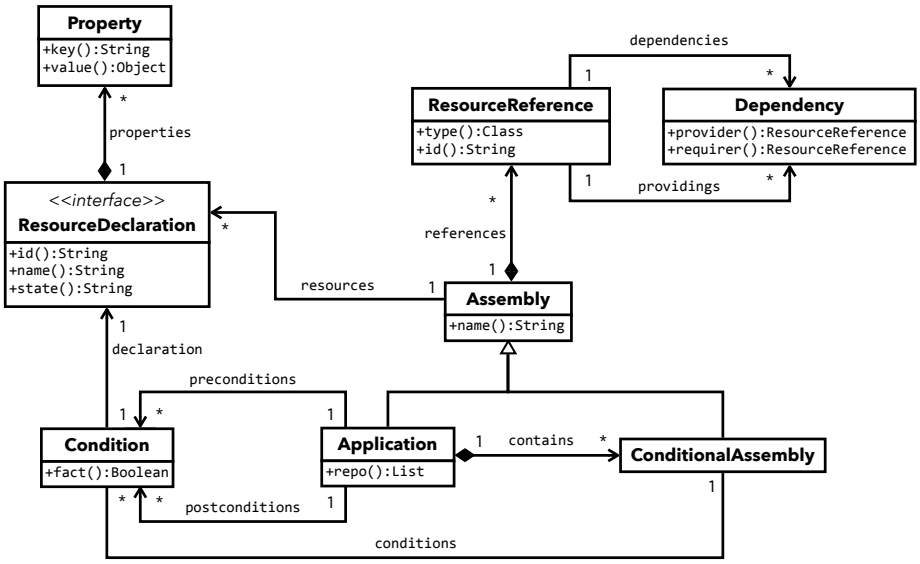


Figure 6.3: Rondo Core Model

The resource concept is modeled using the `ResourceDeclaration` interface. It is a common interface for all types of resources. The `ResourceDeclaration` in-

terface defines `name`, `id` and `state` properties. Different resource types are defined by extending this interface and declaring the additional properties. For example, the interface describing the `Bundle` resource type is shown in listing 6.3. The types of resource declarations are extracted from such interfaces that extend the `ResourceDeclaration`. The implementations of these interfaces constitute actual resource declarations. Resource declaration implementations must build the set of properties the resource describes. It is recommended that resource declaration implementations are developed regarding the resource property types; inquiry, specific and constructive of the conceptual model presented in the previous chapter.

```
1 public interface Bundle extends ResourceDeclaration {

    public String source();
    public String symbolicName();
5   public String version();

}
```

Listing 6.3: Bundle resource type

Here in the example of bundle resource, the `source` is a constructive property, whereas the `symbolicName` and `version` properties are specific properties. The Rondo Core defines and implements resource types commonly found in an execution environment with OSGi and iPOJO: bundles, packages, services, configurations, files, component factories, component instances, etc.

The `Assembly` class regroups a set of `ResourceDeclarations` and allows to define dependencies between them. The *type* and *id* pair of a resource declaration constitutes its unique identifier inside an `Assembly`. This means that an assembly cannot contain two resource declarations with the same type and the same id. A `ResourceReference` contains a *type* and a *id* for referring to a declaration. It allows to define the dependencies between resources. The `Dependency` class represents this relationship and enables navigation of the assembly graph.

The `Application` and `ConditionalAssembly` extend the assembly class. An application being an assembly itself, defines the resources and their dependencies that are unconditional. It contains a set of conditional assemblies, pre-conditions, post-conditions and links to repositories. Lastly, the `Condition` class contains a `ResourceDeclaration` and a *fact*, which is a *boolean* value.

The Rondo Core also provides generic functions for analyzing and manipulating assemblies. These are the functions that are not dependent to resource types such as calculating dependency closures of resources, the `join` operation or the relative complement operation between two assemblies.

Finally, the core model provides a fluent API developed in Java, which constitutes the basis of the Java DSL. It allows to code, in plain Java, application descriptions with re-

source declarations, dependencies, conditional assemblies and pre- and post- conditions. The usage of the Java DSL is detailed in the usage section (see 6.3).

6.2.4 Rondo Deployer

The Rondo Deployer project implements the deployment manager, whose architecture is presented in the section 5.5.2. It receives deployment requests of applications, manages them continuously and executes the deployment process when necessary. The deployer is composed of three types of service-oriented components; the resource processors, the analyzer and the executor. The analyzer and the executor components are singleton and static, meaning that even though they are service-oriented components, they are not replaceable at runtime. But resource processors are dynamic, the Rondo Deployer can be extended with new types of resource processors, taken into account dynamically at runtime. Service interfaces published by the components to the OSGi service registry are noted with the «`service`».

a. Resource Processors

As described earlier in the section 5.5, the deployment managers architecture allows to extend the scope of resource types it can manipulate. A resource processor implements the interactions with resource of a particular type. Stateful resource interactions happen in the `DeploymentParticipant` and the `ResourceMonitor` created by the processor. Any other resource specific function is provided by the `ResourceProcessor` implementation. The diagram 6.4 shows these interfaces and their relationships.

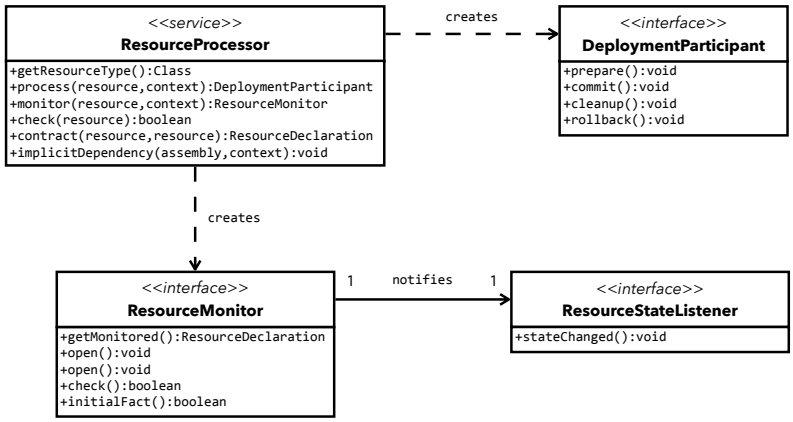


Figure 6.4: Resource Processor Model

The `process` and `monitor` methods of the processor are the factory methods for creating deployment participants and resource monitors. Resource monitors are used by the analyzer for monitoring deployed applications. Deployment participants are coordinated by the executor inside deployment transactions. The implementations of resource processor can `check` a given resource state whether it is fulfilled by the platform or not.

It provides the method `contract` which for given two resources, returns a contracted resource if possible. The resource contraction is possible if two resources are equivalent or one subsumes the other. Finally the resource processor implementation can provide a function for calculating implicit dependencies inside an assembly.

The Rondo Deployer provides implementations of the resource processors for the default resource types defined in the core. These implementations are based on the EveREST framework.

The capabilities of the deployment manager can be augmented by providing custom resource processors for different resource types. These extensions provide the resource declarations for the resource types they manage. They must also implement the `ResourceProcessor` interface and publish it in the OSGi service registry. Other deployer components associate resource processors with the resource type they manage and use them during deployment analysis and execution.

b. Analyzer

The analyzer is a singleton component that analyzes deployment requests and manages deployed applications. It is in charge of continuously monitoring deployed applications and calculating the assembly to deploy. It provides the `DeploymentAnalyzer` interface as an OSGi service. New deployment requests, including new applications or updates for existing ones are introduced into the deployment manager through this service. The diagram 6.5 shows the implementation of the analyzer.

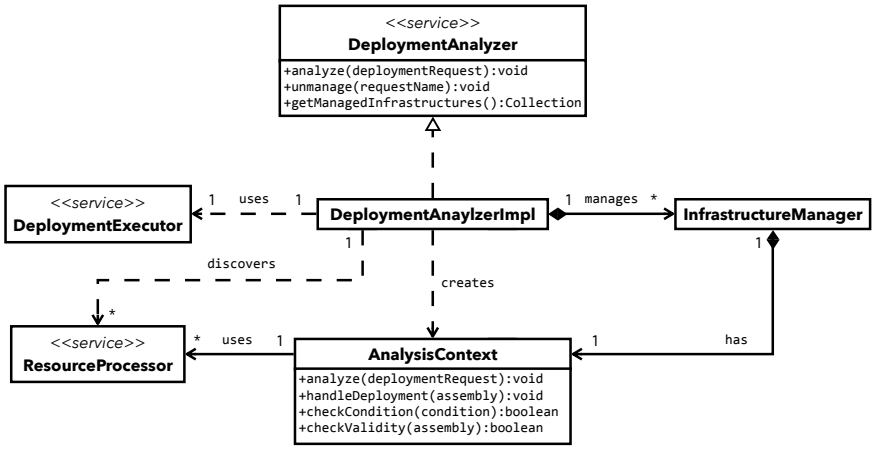


Figure 6.5: Analyzer Model

The analyzer assigns an `InfrastructureManager` to each application to manage its continuous deployment. Infrastructure managers are provided with an `AnalysisContext`, through which they can access necessary information about the current state of the platform. The analysis context is initialized by the anaylzer with the resource processors discovered through the OSGi registry. It provides common methods used during the deployment analysis for checking conditions and validating assemblies.

It can also access the `DeploymentExecutor` service for requesting the deployment of an assembly (presented in the following part of this section).

These functions provided by the analysis context are used in the analysis of the infrastructure managers for new deployment requests and for adapting running applications. The `InfrastructureManager` is in charge of conducting analysis operations for the application they manage. On one hand, it evaluates the deployment requests, for the first time an application is to be deployed and later for the pushed updates. Each deployment request contains the application management information it targets.

On the other hand, once deployed, the infrastructure manager oversees the monitoring mechanism to trigger adaptations when necessary. The implementation of the `InfrastructureManager` is depicted in the diagram 6.6.

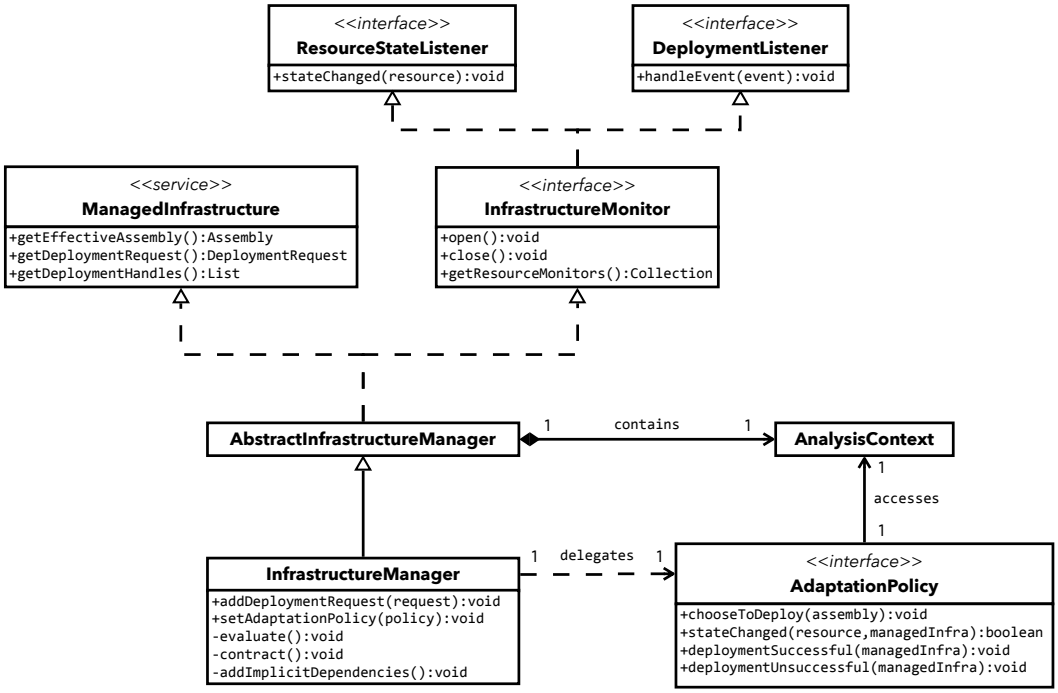


Figure 6.6: Infrastructure Manager Model

The analyzer component starts the infrastructure manager with an initial deployment request and later on with each new deployment request, analyzer redirects it to the corresponding infrastructure manager. The infrastructure manager holds the information on the effective assembly deployed and the chronological order of the deployment handles (also presented in the following part of this section).

The infrastructure manager starts and maintains the monitoring mechanism for the deployed application. It creates the `ResourceMonitors` using appropriate resource processors and adds itself as listener for resource state changes.

Together with the deployment request, it is possible to set a custom adaptation pol-

icy for each application by providing a class implementing the `AdaptationPolicy` interface. The adaptation policy has access to the `AnalysisContext` of the managed infrastructure. It implements the `chooseToDeploy` method to decide on which contained assemblies are going to be chosen for deployment. The infrastructure manager delegates deployment events and resource state changes to the adaptation policy, for that the policy can decide whether to trigger a deployment process. The analyzer already provides a default implementation of this interface to be used when no custom policy is set.

c. Executor

The executor component implements the planner module of the reference architecture. It is a singleton component that implements the `DeploymentExecutor` interface and publishes it in the OSGi registry. The main method of this interface, `handle`, takes an assembly to deploy as input, plans and executes its deployment. The diagram 6.7 shows the architecture of the executor component.

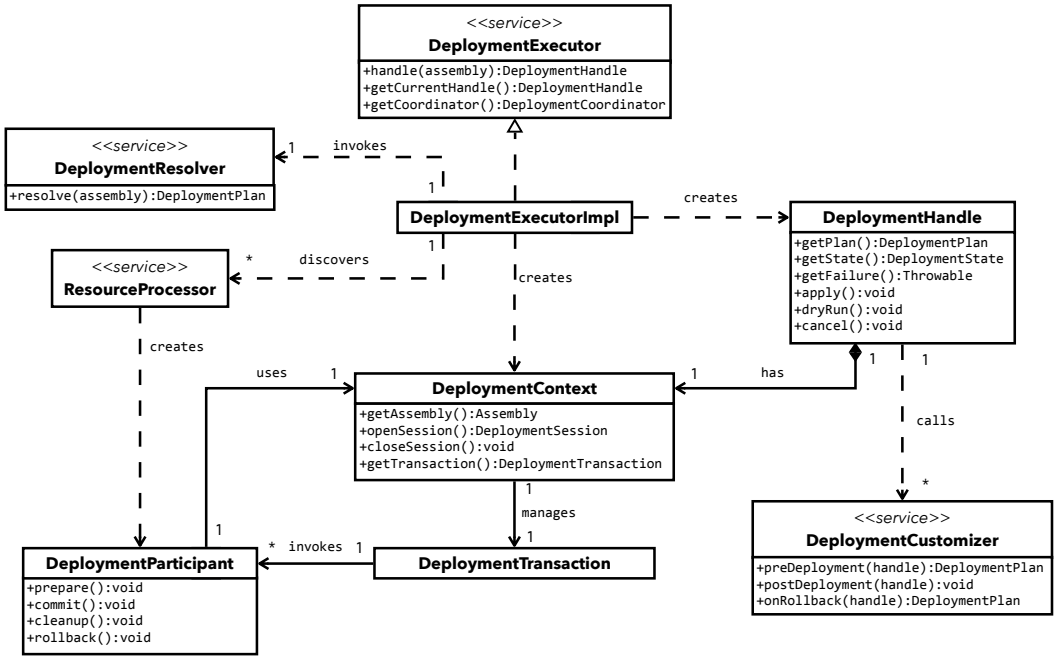


Figure 6.7: Executor Model

The handle method returns a `DeploymentHandle` immediately after creating the deployment plan for the assembly. As discussed earlier there are many possible implementations for creating the deployment plan. Therefore the `DeploymentResolver` implementations are separated from the executor. The executor depends strictly on a resolver service it discovers from OSGi registry for resolving deployment plans.

The executor maintains the deployment queue, which ensures that there is only a single deployment process executing on the platform. If the resolver returns a deploy-

ment plan, the executor enqueues the plan for deployment. A single thread handles the assemblies on the queue one by one in the order of arrival.

The `DeploymentHandle` represents a specific deployment process. It is created with the deployment plan, the deployment context and a `DeploymentCustomizer` if available. It allows to start the deployment proces or cancel an ongoing one. It also allows to make a dry run, which tests only the prepare phase of the deployment. The handle of a deployment enables the introspection of the process, offering the deployment state and the failure, if the deployment failed. Possible states for a deployment process are as the following:

- **`CREATED`**: Deployment enqueued, but not yet started.
- **`DRYRUNNING`**: Deployment running on dry mode. The execution won't affect the platform and prepared resources will be cleaned up.
- **`RUNNING`**: Deployment created and started running.
- **`UNSUCCESSFUL`**: Deployment finished but was unsuccessful. The `getFailure` method returns the root reason of failure as a `Throwable`.
- **`SUCCESSFUL`**: Deployment finished and was successful.

The state changes are notified to the `DeploymentListeners` that can be registered to the deployment handle. For example, `InfrastructureManagers` are registered to the deployment handles they requested.

The `DeploymentCustomizer` is a service discovered from the OSGi service registry by the executor. It is called by the handle before, after and on rollback of deployment. The customizer can return a modified deployment plan on the `predeployment` callback. Similarly, `onRollback` callback can return a deployment plan to recompensate on the failure and continue the deployment with the alternative plan.

The `DeploymentContext` manages the deployment session as well as the deployment transaction. The deployment handle prepares the deployment transaction by creating `DeploymentParticipants` from corresponding processors and adding those to the transaction according to the deployment plan. Deployment participants have access to the deployment context for using configurations of the particular deployment. They are coordinated by the `DeploymentTransaction`, as explained before in the reference architecture 5.5.2.c.

6.2.5 Resolvers

As explained above, there are many possible algorithms to calculate the deployment plan, thus there are different implementations of the `DeploymentResolver` interface. A

resolver is in charge of calculating the deployment plan but also reporting that a plan cannot be calculated in case of a cyclic assembly. Rondo proposes three functional implementations of the resolver, each with different characteristics; breadth-first resolver, depth-first resolver and the topological ordering resolver. The following chapter evaluates the performances of these different algorithms.

a. Breadth-first Resolver

The first resolver implementation applies a breadth-first search (BFS) on the assembly graph for constructing the deployment plan. It implements the BFS iteratively from bottom of the tree, up to roots. Starting from resources without dependency, in each iteration, resource declarations that all the requirements are already visited are added to the deployment plan.

This resolver is the only one that creates parallel resource declarations to the deployment plan. However, it does not provide detailed error information if the assembly is cyclic.

b. Depth-first Resolver

The second resolver implementation applies a depth-first search (DFS) on the assembly graph. The search starts from an arbitrary resource declaration and drills down to its dependencies recursively. It creates the resource order according to the ***preordering***, i.e. the order they are visited by the DFS. If all the dependencies are already in the deployment plan, then the visited resource is added next in the plan. If there are still resources that are not visited, the search continues on by choosing one of them and recursively searching its dependencies.

Notice that this implementation choses arbitrarily resource to search in depth, meaning that the order of resources is arbitrary. This is a weakness for the implementation, because it is not deterministic. For its advantage it can detect cycles giving more detail about the detected cycle.

c. Topological Ordering Resolver

The last resolver implementation applies a topological sort algorithm. The resolver uses JGrapht[3] library for detecting cycles and calculating the topological order. The topological ordering creates a ***reverse postordered*** resource list, meaning that resources in the deployment plan are ordered in the inverse of the last visit order during a DFS. Instead of handling resources in arbitrary order like the DFS, the topological ordering can lexicographically order the dependencies according to ids of resource declarations.

Using the JGrapht library, the detection of cycles is more detailed. The library offers a tool for detecting all of the cycles in a given DAG. With lexicographically ordered topological sort, the deployment plan calculcated is deterministic.

[3] http://jgrapht.org/

6.2.6 Rondo Cloner

The Rondo Cloner is an reverse-engineering tool, which allows to create the deployment descriptor from an already executing OSGi platform. Using the graph of resources represented by EveREST framework it creates the Rondo model as an assembly and writes the deployment descriptor into a file with the Java DSL.

The tool is called ***cloner*** because the descriptor written into the Java source code can then be compiled and deployed to another platform in order to reproduce the cloned platform (see figure 6.8).

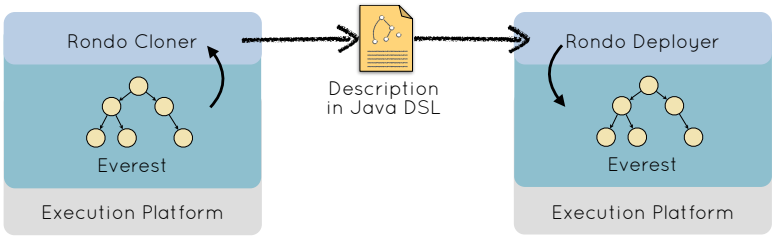


Figure 6.8: Rondo Cloner

The cloner follows the extensible architecture of Rondo and EveREST. Resource processors can provide a `ResourceWriter` interface as service which writes the declarations of different types of resources into descriptor source code. As example follows, the cloner provides writer for the core resource types, and discover other resource writer from the OSGi registry.

6.3 Usage

This section explains the installation and usage of Rondo tools, as well as the examples for coding deployment descriptors using Rondo DSLs. It also details how to develop resource processors for custom resource types.

6.3.1 Installation

As expressed previously, Rondo is executed on OSGi platforms. It is developed using the iPOJO component model and depends on the EveREST framework. The basis for an OSGi platform that is configured with Rondo tools contain following bundles.

- **Apache Felix iPOJO**: iPOJO is needed both for EveREST and Rondo bundles as they are designed and developed as several service-oriented components, presented in the previous section.

- **EveREST Bundles**: EveREST bundles include the ***EveREST-core***, ***EveREST-osgi***, ***EveREST-ipojo*** and any other EveREST domain. EveREST OSGi domain has optional dependencies to OSGi Configuration Admin and Deployment Admin Package services for representing these domains.

- **OSGi Event Admin**: The OSGi Event Admin is used for delivering EveREST event for resource notifications. A stable and compliant implementation such as Apache Felix Event Admin is recommended for use.

- **Rondo Bundles**: Rondo bundles necessary for the deployment manager are the ***core***, the ***deployer*** and a ***resolver***. The core bundle provides the model and some abstract classes for developing new resource types. The deployer bundle provides the components of the deployment manager, resource processors for standard EveREST domains and the abstract classes useful for developing other resource processors. A resolver implementation is required for the deployment system to function. In addition to those, the ***command*** bundle provides OSGi console commands that integrates to the Apache Felix Gogo Runtime. These commands serve to introspect the deployment system, managed applications and clone the platform. Finally, a deployment customizer can be provided per platform for customizing the deployment process.

- **Rondo Extensions**: Rondo can be extended with resource processors for different resource types. The ***system*** extension is an example for Rondo extensions, which depends on the EveREST domain with the same name.

- **Rondo Cloner**: The ***cloner*** component is packaged in a separate bundle from the deployer. It only depends on the ***core*** and it is not required by the deployment manager.

- **Rondo Groovy DSL Bundles**: The ***Groovy DSL*** support is provided by the Groovy Script Deployer bundle. It requires the Groovy Runtime to be installed on the OSGi platform. The usage of the Groovy DSL is explained later in this section.

- **URL Handlers**: The resource processors that construct resources on the platform usually need to fetch artifacts (bundles, jars or any other files) from remote filesystems or repositories. As explained earlier, Rondo does not deal with resolution of artifacts from software repositories. Instead, it delegates this to the resource processors capabilities to fetch artifacts. An example to this by relying on the URL handlers of OSGi platform. For example, OPS4J Pax Url Handler[4] for Apache Maven URLs resolves the `mvn://..` links inside given Maven artifact repositories.

In addition to the installation of bundles to the OSGi platform, Rondo does not need any other configurations. Once installed Rondo Deployer is ready for accepting deploy-

[4] http://github.com/ops4j/org.ops4j.pax.url

ment requests written in Java or Groovy DSLs. Following sections explain how to develop deployment descriptors using these DSLs.

6.3.2 Java DSL

The Java DSL consists of a ***fluent API***, provided by the Rondo core. The concept of fluent API is described by Martin Fowler and serves, among other things, to write a type of DSL called ***Embedded DSL***. An Embedded DSL is written using another programming language for leveraging the constructs and also the already existing tools of that programming language.

Rondo Java DSL is embedded inside Java. It is based on the fluent API provided by the Rondo Core for creating application descriptions with assemblies and resource declarations. Despite of being a based on a fluent API, the description language is declarative. The descriptions written using this DSL are compiled using standard Java compiler and need only the Rondo Core in the classpath. The compilation phase grants the syntactic checking of Java to the descriptor. It also enables establishing a build process with other model checking mechanisms for consistency and coherence of the coded descriptor.

Application descriptions written in Java DSL are annotated with `@Application` or `@Infrastructure` annotations, compiled and packaged into OSGi bundles. The deployer provides a mechanism for processing bundles, extracting the annotated classes, creating the descriptions and transferring those to the analyzer as deployment requests. Following are the portions of code for illustrating the usage of the Rondo Java DSL.

```
1
@Application(id = "example-application", version = "1.0.0")
public class ExampleApplication {

5    AssemblyImpl app = assembly();
```

Listing 6.4: Application description - Java DSL

In this example the ExampleApplication class is annotated with the Application annotation, specifying the identifier and the version of the application. Then the assembly model is initiated with the `assembly` method.

```
    app.resource(pckg("config-admin")
2           .name("org.apache.felix.config.admin")
            .version("1.2.6"))
       .resource(bundle("icasa-bundle")
            .symbolicName(name)
6           .version(bundleVersion)
            .state("ACTIVE")
            .source("mvn:fr.liglab.adele.icasa/"+name+"/"+version))
```

Listing 6.5: Resource Declaration - Java DSL

Once the assembly model is initiated, the resources can be added by calling `resource` method, which takes a ResourceDeclaration as parameter. In this example resources of *configuration*, *package* and *bundle* are added to the assembly.

```
            .resource(Bundle.class,"icasa-bundle")
                .dependsOn(Package.class,"config-admin")
            .resource(Configuration.class,"conf")
                .dependsOn(Bundle.class,"icasa-bundle");
```

Listing 6.6: Resource Dependencies - Java DSL

Dependencies between resources are added to the assembly by calling its `resource` method, which in this case takes a resource type and a resource id. It returns a `ResourceReference`, to which dependencies are added calling the `dependsOn` method.

```

    app.when(condition(configuration()
        .pid("org.ops4j.pax.url.mvn"))
            .isTrue())
        .then("appfragment",appFragment());
}
```

Listing 6.7: Conditional Assembly - Java DSL

This illustrates how to add conditional assemblies to the assembly. The `when` method allow specifying the condition set of the conditional assembly. Conditions are created using the `condition` method, which creates a condition with the given `ResourceDeclaration`. The condition designates the fact of the condition by `isTrue` or `isFalse` methods. Finally, the `then` method adds the assembly in its parameter to the main assembly.

```
public Assembly appFragment(){
    return assembly()
            .resource(zigbeedevice("discovery"))
            .resource(zigbeedevice("factories"))
            .resource(zigbeedevice("importer"))
            .resource(zigbeedriver("api"))
            .resource(zigbeedriver("impl"));
}
```

Listing 6.8: Method returning an assembly - Java DSL

This example shows the `appFragment` method called in the conditional assembly of the previous example. The descriptor code is still standard Java, which allows including other libraries to the code, possibly referring to other assembly descriptions. In this case, the method returns an assembly including five resources that are constructed using different methods.

6.3.3 Groovy DSL

The descriptors written in Java DSL need a compilation phase and are introduced to the deployment manager as OSGi bundles. Groovy DSL, on the other hand, is written as scripts and compiled directly on the execution platform. The syntax of the Groovy DSL is detailed previously in the section 5.6.

Scripts written in Groovy DSL are saved into files with `.rondo` extension. The ***Groovy Script Deployer*** component handles these files, executes the script, which creates deployment requests. By default, the script deployer monitors a directory in the local filesystem for changes and each time there is a change, the deployment description is reconstructed and transferred to the ***analyzer***. Similarly to the Java DSL, these scripts leverage Groovy language. It is possible to create methods and assign variables. The following example shows a simple application written in Groovy DSL.

```
package fr.liglab.adele.rondo

def infra = assembly "icasacommon",false,false, {
  resource icasacommon { bundle
    'bundle-symbolic-name' "fr.liglab.adele.icasa.common"
    state "ACTIVE"
    'bundle-version' "1.2.6.SNAPSHOT"
    source "mvn:fr.liglab.adele.icasa/common/1.2.6-SNAPSHOT"
  }
}

application id:"example-appliction", name:"example",
  version:"0.0.1", vendor:"ozan",{

  with infra
  resource dashboard { bundle
    'bundle-symbolic-name' "fr.liglab.adele.icasa.dashboard.web.instance"
    state "ACTIVE"
    'bundle-version' "1.2.6.SNAPSHOT"
    source "mvn:fr.liglab.adele.icasa/dashboard.web.instance/1.2.6-SNAPSHOT"
  }
```

Listing 6.9: Application example - Groovy DSL

Moreover, the script deployer handles together different scripts that have the same package name.

6.3.4 Resource Processor Development

The deployer project already provides resource processor implementations for common resource types in the context of this work. As expressed earlier Rondo tools are extensible with new resource types. Development of these resource processors is crucial for the characteristics guaranteed by the deployment process, notably the determinism and idempo-

tence. Therefore, it is also necessary to help developers program new resource processors and integrate those to development tools provided by Rondo. This section demonstrates resource processor development by presenting two use cases: the ***fragment*** and ***parallel*** resource types.

a. Fragment Resource Processor

Fragments are special type of bundles in the OSGi specification. They are attached to one or more *host* bundles, as part of the package resolution. The framework appends definitions of the fragment to the host bundle, before the resolution of the host. Fragments are therefore treated as part of the host when loading classes or accessing other Java resources. Contrary to the standard bundles, fragments are never *activated* but only *resolved* if and only if they are attached to a host.

This mechanism enforces a constraint on the deployment of fragments. The host of the fragment must already be installed before the fragment, therefore the fragment depends on its host. But even though it is not explicit, in most of the cases the host bundles resolution depends on the existence of fragments. Indeed this results in a cyclic dependency system between fragments and their host bundles. A general practice for deploying fragment bundles is to install them in two steps. Host bundles are installed without resolving before the installation of fragments. Then once the fragments are installed and attached to the hosts, hosts are resolved and started.

The cyclic dependency poses a problem for the deployment plan resolution in Rondo. But above all the previous general practice is not applicable neither. Notice that the Rondo deployment descriptors declare the expected, ***final*** state of resources, not the deployment process. The deployment process and the actions it contains are inferred from that target state. Therefore describing the previous ***process*** results in creating two resource states for the host bundle such as `INSTALLED` and `ACTIVE`, which are obviously conflicting.

This is why the development of ***fragment*** resource processor is an important use case for understanding the expected state described by resource declarations and the idempotence of resource processor implementations. The solution implemented in the Rondo Deployer project is presented here.

The fragment resource declaration is similar to the one of bundles, describing the *symbolic name*, the *version* the *state* and the *source* of the fragment. In addition to that, fragment declaration includes a declaration of the host bundle. The following example shows an example of fragment declaration.

```
        fragment("slf4j.simple")
3           .source("mvn:org.slf4j/slf4j-simple/1.6.6")
            .symbolicName("org.slf4j.simple")
            .version("1.6.6")
            .state("RESOLVED")
7           .host(bundle("slf4j-api")
```

```
                .source("mvn:org.slf4j/slf4j-api/1.6.6")
                .symbolicName("org.slf4j.api")
                .version("1.6.6"));
11
    }
```

Listing 6.10: Fragment Declaration Example

The deployer project provides the `AbstractDeploymentParticipant` abstract class for facilitating deployment participant development. The abstract class gives access to the involved resource declaration and the deployment context. The deployment participant for fragment resources extend this class and implement the methods for participating to the transaction. The following resumes the implementation of these methods:

- `prepare`: The prepare method first checks whether the given resource declaration is well-formed or not. In case of fragments, this is done for both the fragment and the host definition. Then it checks if in the current state of the platform, one or more resources correspond to this declaration. If found it backs up the state representation of these resources. In case of fragments, the state of the fragment and the host bundle, and their source (using the *bundle-location* property) are backed up. This is one of the reasons why deployment participants are stateful objects. If any resource exists corresponding to the declaration and the given declaration is not constructive, the prepare throws an exception. If the declaration is constructive, it prepares necessary files or configurations for constructing the resource in the next phase. For fragments and bundles, the `prepare` method downloads the bundle file and checks if the manifest information corresponds to the given declaration.
- `commit`: The commit method is in charge of making the changes on resource states, if necessary, and checking if these changes are applied. Fragment deployment participant first makes sure the host bundle is at least installed, and then proceeds with installation of the fragment bundle. If any bundle installations are needed it uses the files and configurations prepared in the prepare phase. Finally, it makes a last check of the state of resources.
- `cleanup`: The cleanup method is called if the transaction fails after or during the prepare phase. In case of fragments, it makes sure the downloaded and prepared files are deleted.
- `rollback`: The rollback is called if the transaction fails during the commit phase. It makes best effort to restore the backed up state of resources. In case of bundles and fragments it reinstalls the bundle to the OSGi with the backed up file, if necessary.

The deployment participant code involves a lot of error handling. Fragment resource processor leverages the EveREST framework for gathering the state and manipulating the

bundles. This results in uniform interfaces and exception handling in the implementation of deployment participant.

For the implementation of resource monitors, another abstract class, `AbstractResourceMonitor` is provided by the deployer project. The implementations that extend this class implement the `open`, `close` and `check` methods. As explain earlier in this chapter, `check` method is also implemented by the resource processor to check whether a declaration is true for the current state of the platform. The `open` and `close` methods start and end the monitoring of the resource state. The resource monitoring implementation must observe the resource state and calls the `stateChanged` method provided by the abstract class to notify the state change. For some resource types the push notifications can be obtained from the platform, but for others the resource monitor should poll the resource periodically and for controlling if the state has changed. For instance, the EveREST framework sends notifications for state changes of any bundle type, including fragments. But such notifications are not available for system properties in Java Runtime Environment.

b. Parallel Resource Processor

Depending on the resolver that constructs the deployment plan, Rondo deployer is capable of running resource participants in parallel. This is made possible by implementing a ***parallel*** resource processor. A parallel resource declaration is constructed with several other declarations. The resource processor holds a thread pool for executing participant tasks in parallel. The size of the thread pool can be configured according to the platform machine, for example related to the number of processors. At each transaction phase, the parallel deployment participant coordinates the participants it contains and invokes their corresponding actions (see figure 6.9). The consequences and advantages of executing some of the deployment actions in parallel are evaluated in the following chapter.

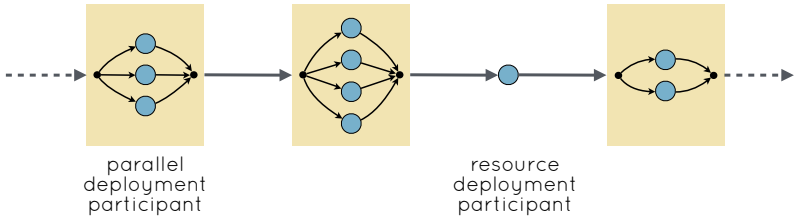


Figure 6.9: Parallel Deployment Participants

6.4 Conclusion

This chapter presented the important points about the implementation of the reference architecture, the Rondo deployment tools and the EveREST context representation framework. Both Rondo and EveREST are available as open source in GitHub respectively at https://github.com/AdeleResearchGroup/Rondo and https://github.com/ow2-chameleon/EveREST.

The EveREST project implements an extensible framework for representing context, adapted for dynamic environments. This chapter shows how EveREST framework can be extended for representing different types of context entities from the platform. Using the context representation provided by EveREST, Rondo project implements various tools for deployment facilities. This chapter studies in detail how these tools are implemented and arguments different implementation choices. These tools are equally extensible for taking into account different resource types available on a deployment site. This chapter also presents guidelines for extending the capabilities of Rondo tools by implementing resource processors.

Rondo and EveREST serve in the following chapter for testing and validating the contributions of this thesis.

The following table (see table 6.1) indicates the lines of code and lines of test code in each of the projects developed for Rondo deployment tools.

Table 6.1: Lines of code in Rondo project

Project		LOC	LOTC
Rondo Core	Model	136	329
	Implementation	1219	
	Utils	135	
Rondo Deployer	Analyzer	494	511
	Executor	443	
	Transaction	278	
	Resource Processors	2826	
	Utils	1821	
	Other	516	
Rondo Cloner	Cloner	643	108
	Writers	385	
Rondo Command		270	–
Rondo Groovy Lang		550	44
Resolvers	Simple	59	92
	Recursive	48	
	Jgrapht	53	
Extensions	System	173	48
	Framework	289	65
Descriptor Examples	Shell	776	–
	iCASA	755	
	Application	42	
	Infrastructure	42	
Total		**11648**	

Chapter 7

Validation

" To understand is to perceive patterns. "

— Isaiah Berlin

Contents

7.1 Introduction

The previous chapter describes the implementation details of the context representation framework, Everest; and the set of deployment tools, which implement the main contributions of this thesis, called Rondo. The previous chapter also gives instructions on how to use principal functionalities of Rondo and how to extend it with different resource processors.

This chapter provides validation for the deployment capabilities of Rondo framework. First, in the next section, different resolver implementations are tested by calculating deployment plans for different assemblies. Secondly, the chapter continues by presenting different use cases in which the Rondo deployment manager is tested. Presented tests and evaluations focus on validating four properties of Rondo framework:

- ***Performance and overhead acceptability:*** The comparative acceptability of deployment process performance and the overhead on the idle platform.
- ***Deployment versatility:*** The ability to be used in different deployment scenarios, including deployment of platform technical services and applications.
- ***Error handling and diagnosis:*** The reaction and flexibility offered in face of errors during the deployment process.
- ***Dynamic adaptability:*** The deployment manager's ability to adapt the deployed applications dynamically according to the defined variability.

7.2 Resolver Evaluation

In the section 6.2.5 of the previous chapters presented three different implementations of resolver components[1]. To recall briefly, the deployment plan designates the order of which the deployment actions are applied by the deployment manager. The calculation of this deployment plan is therefore a key step in the deployment process. The resolver is in charge of resolving the graph of the assembly and producing a deployment plan as a result. This first section presents the results of the performance tests performed in order to evaluate and compare these different algorithms for deployment plan resolution. These performance tests are executed on a MacBook Pro featuring 8 GB of RAM and an Intel 2.53Ghz Core 2 Duo processor. It runs OS X 10.9.4 64-Bit operating system and Java HotSpot version 1.8.0_05, 64-Bit Server virtual machine.

The resolver performance tests evaluate the execution time of deployment plan resolution of different assemblies. The set of test assemblies is a mixture of assemblies that are used for deployment; noted by *Shelbie*, *iCASA* and *iCASA & Wisdom* and other assemblies

[1] components which provide resolver service for calculating the deployment plan of assemblies

that are obtained from generated graphs. For instance, OW2 Shelbie[2] console is a textual shell implementation for OSGi platforms. iCASA and Wisdom frameworks are introduced later in this chapter. The assemblies *Graph-150*, *Graph-300*, *Graph-487* and *Graph-800* are generated randomly conforming to directed acyclic forest graphs. They are generated by determining a fixed node size, in order to evaluate the performance changes of resolvers.

The table 7.1 shows the properties of these assemblies and execution times of three resolvers; depth-first search (DFS), topological sorting (Topsort) and breadth-first search (BFS). The mean and median of execution times, measured in milliseconds, are calculated as the result of 1000 iterations of resolution.

Table 7.1: Deployment plan resolution comparison

Assembly	# of nodes	# of edges	DFS		Topsort		BFS	
			median	mean	median	mean	median	mean
Shelbie	54	94	0	0.13	3	3.09	0	0.5
Graph-150	150	250	0	0.32	7	8.20	2	2.10
Graph-300	300	453	1	0.64	12	13.17	4	5.03
Graph-487	487	680	1	1.06	21	23.43	7	8.36
iCASA	487	879	1	1.20	23	24.57	7	7.33
Graph-800	800	1415	2	2.02	46	52.73	19	20.30
iCASA & Wisdom	1036	2177	3	3.26	78	91.05	39	44.52

The chart in the figure 7.1 summarizes the comparison of three resolver algorithms.

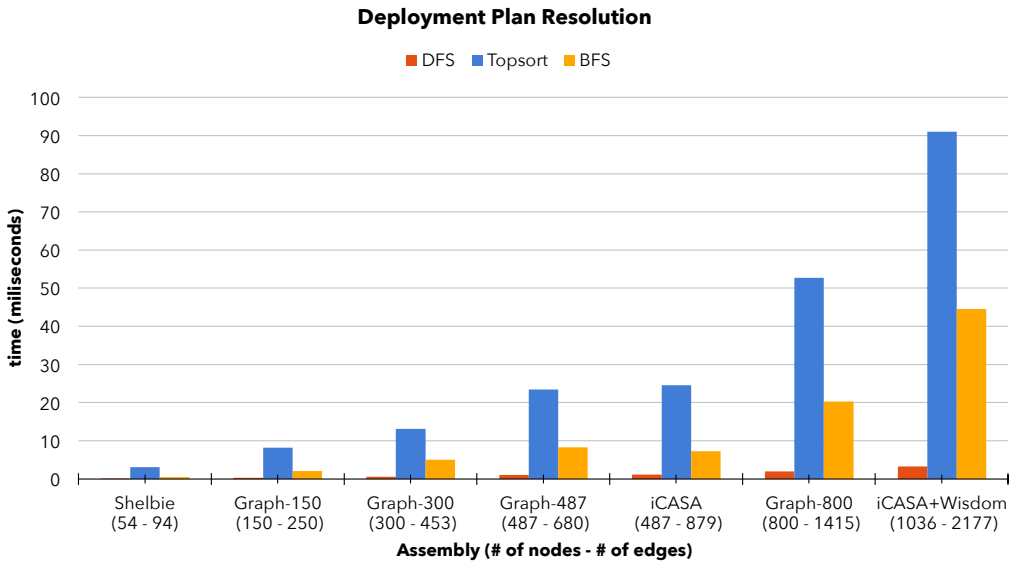


Figure 7.1: Deployment Resolver Performance Comparison

All three of the algorithms produce acceptable calculation times, even for large assemblies containing thousand nodes. But otherwise, they fare differently according to

[2]OW2 Shelbie: http://shelbie.ow2.org/

the properties of the assemblies they receive. The DFS resolver implementation is by far the fastest among them. However, it produces sequential deployment plans. The BFS resolver, which produce parallel deployment plans, is slower than the DFS. The resolution times increase linearly with relation to the size of the assembly. The Topsort resolver implementation uses an external library, Jgrapht. This explains the higher execution times, even for the smallest size assemblies, because the assembly models are transformed into the Jgrapht graphs before running the topological sorting. But once this overhead is accepted, it produces acceptable resolution times.

Despite its secondary place in assembly resolution, the following tests mostly use the BFS resolver implementation. The foremost reason for this choice is its ability to produce deployment plans that enable parallel deployments. As the following section shows, executing deployment actions in parallel greatly improves the execution time of deployment processes. Furthermore, in platforms where parallel execution of deployment actions is not favored, the BFS or Topsort resolver implementations are still a viable choice. In the remaining sections of this chapter, the comparison between parallel and sequential deployment plans are displayed through BFS and Topsort resolvers.

7.3 Performance Evaluation

The goal of this section is to evaluate the performance acceptability of Rondo with comparison to other currently used deployment methods on OSGi™ platforms. This evaluation comprises a comparison of metrics for the deployment platform and the deployment process. The metrics for the deployment platform measures the adoption cost of the deployment tool. These metrics are:

- ***Start-up duration:*** The time it takes from the launch event of the deployment platform until all of the resources are initialized and fully operational.
- ***Idle memory consumption:*** The memory consumption of the platform while it is not active, i.e neither a deployment process nor any application is executing on the platform.

On the other hand, following metrics measure the performance of the deployment process:

- ***Deployment process duration:*** The time it takes from the request for deploying a test application until all of the resources of this application are fully operational.
- ***Deployment process CPU consumption:*** The maximum CPU consumption during the deployment process.

7.3.1 Test Application

The deployment process is evaluated by deploying a test application that can execute on all of the platforms. This test application is a simple service-based application that is very common on OSGi platforms. The implementation uses iPOJO for defining components, instances and providing OSGi services. It is composed of a ***library*** module, an ***API*** module, a ***support*** component, a ***server*** component and a ***client*** component. The application includes a single non-conditional assembly that is showed in the graph depicted in the figure 7.3.1. To give an order of scale, this assembly declares 20 resources and 23 dependencies.

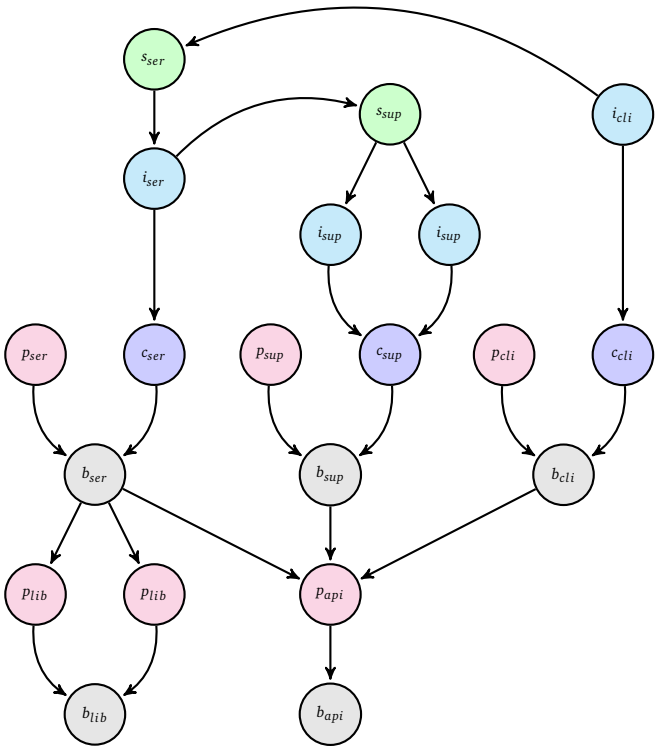


Figure 7.2: Assembly Graph of the Test Application

The ***API*** module contains the service contracts, which are defined separately for a better decoupling between client, server and support components. The ***server*** module depends on the ***library*** for providing the main service of the application, with the help of at least one or multiple ***support*** services. This exemplifies two kinds of dependencies in the OSGi platforms, package level-dependency and service-level dependency. The behavior of the server in delivering its service changes according to the number of support services it has in its disposition. Note that in the assembly graph, two initial instance resources are declared for the support component. However, the server instance depends on the existence of a support service. Lastly, the ***client*** component requires a service provided by the server to be active and calling the server.

7.3.2 Tested Platforms

As explained before, the performance acceptability tests involve comparing multiple platforms with different deployment methods. Following are the brief descriptions of the platforms tested in the context of this comparison:

Baseline framework – OW2 Chameleon Core: The test baseline is an OSGi framework based on the Apache Felix implementation, version 4.4.0, structured with the open source project OW2 Chameleon Core [3]. It simplifies the distribution of customized OSGi based platforms. It includes a number of core technical services such as interactive console, logging backend and OSGi Configuration Admin support. Other platform configurations in this list are built upon this baseline platform.

Deployment Admin Package: The Deployment Admin Package is a specification first included in the OSGi specification version 4.1 [OSGi Alliance 2007] for managing runtime configurations of an OSGi platform. It is mentioned and described several times in this document. The platform configuration for testing deployment admin package deployment includes an implementation of the Deployment Admin service and the Autoconf resource processor that serves processing OSGi Configuration Admin configurations from files. These implementations are open source and provided by akquinet AG[4].

Apache Felix File Install: The Apache Felix File Install is an utility for watching directories in the filesystem for managing the runtime configurations of an OSGi platform. The content found in the watched directory constitutes the configuration of the platform. It is capable of processing and installing OSGi bundles and creating Configuration Admin configurations. File Install also allows adding new listeners on the watched directory for handling custom artifact types. Mostly because of its simplicity File Install is a widely used tool for conducting deployments in OSGi platforms. This test configuration includes the Apache Felix File Install version 3.2.6.

Everest: The context representation framework Everest doesn't conduct deployment processes but is included in this test in order to evaluate the footprints of Everest and Rondo frameworks.

Rondo - Topsort: The platform configuration includes Rondo deployment manager with the Topsort resolver implementation and the system extension, without the Groovy Language extension.

Rondo - BFS: The last platform configuration is same as the previous one, except that it uses the BFS resolver implementation. This serves to recognize the effect of parallelization of deployment actions on deployment process duration.

[3] OW2 Chameleon - Core: http://ow2-chameleon.github.io/core/snapshot/
[4] Deployment Admin: https://github.com/akquinet/osgi-deployment-admin

Note that these platforms are instrumented for being able to execute measurement tests. For measuring start-up and deployment durations, platforms are instrumented in order to generate and capture events that signal the start and end of the durations. Start-up times are measured by modifying the baseline framework for registering the events of platform start and of bundle and service stability. Likewise, the durations of deployments are registered using an iPOJO component that tracks an event for the start of the deployment and measures the time until the bundle and service stability. For low level metrics such as CPU consumption and memory usage, YourKit[5] profiling tool is used.

7.3.3 Test Results & Remarks

The table 7.2 presents the results of conducted comparative tests. All the performance tests are executed on a MacBook Pro featuring 8 GB of RAM and an Intel 2.53Ghz Core 2 Duo processor. It runs OS X 10.9.4 64-Bit operating system and Java HotSpot version 1.8.0_05, 64-Bit Server virtual machine. It is useful to recall that this JVM uses G1 garbage collector [Detlefs 2004] as default and does not have a PermGen memory space.

In the context of these tests prepared platforms are launched 10 times (following a 3 times warm up period) in order to measure the start-up durations. The idle memory consumption shown is the sum of heap and non-heap used memory. The profiling tool is used to measure the memory consumption at the 1-minute mark from the launch and initialization of the framework, when there is no activity after a full garbage collection. Then the test follows by deploying the test application and measuring the time between the start event of the deployment and the moment there is no more bundle and service activity in the framework. In the case of Rondo, the default deployment customizer of Rondo provides the beginning and the end of the deployment process. During the deployment process, the CPU consumption is measured using the profiling tool by tracing the CPU percentage with 1-second intervals.

Table 7.2: Test application deployment comparison

Platform	Start-up Duration (ms)	Idle Memory (Mb)	Deployment Duration (ms)	Max. CPU %
Baseline - Chameleon Core	2694	35.87	–	–
Deployment Admin Package	2978	37.74	1034	23
Apache Felix File Install	2867	36.49	1063	51
Everest	3549	40.65	–	–
Rondo - Topsort	5876	46.85	1094	65
Rondo - BFS	5885	45.89	899	72

The results show that start-up durations and memory consumptions follow the complexity of the deployment method. The Deployment Admin service and the Apache Felix

[5]YourKit : http://www.yourkit.com/

File Install introduce small overhead on start-up and memory compared to the baseline. The overhead of Rondo (combined with Everest framework) is higher but still would not be significant in a larger system.

The deployment durations and CPU consumption of tested solutions are close but comparable. The chart in figure 7.3 shows the distribution of deployment durations in milliseconds. The deployment process of Deployment Admin service is slightly faster and consumes less CPU than the File Install and Rondo with Topsort resolver. This can be explained by the process of bundle activation. Both the Deployment Admin and the File Install employ a two-step process to handle OSGi bundles. First they call the OSGi framework to install the bundles that constitute the test application. At each bundle install, the OSGi framework analyzes the dependencies of bundles and tries to resolve them. Once all bundles are installed, Deployment Admin proceeds by activating all resolved bundles. File Install, however, gets notified each time a bundle is resolved, and tries to activate the bundle. This explains why all deployment experiments with Deployment Admin result concentrated times. With comparison, the installing and activation of File Install depends on the order the files are copied to the watched directory.

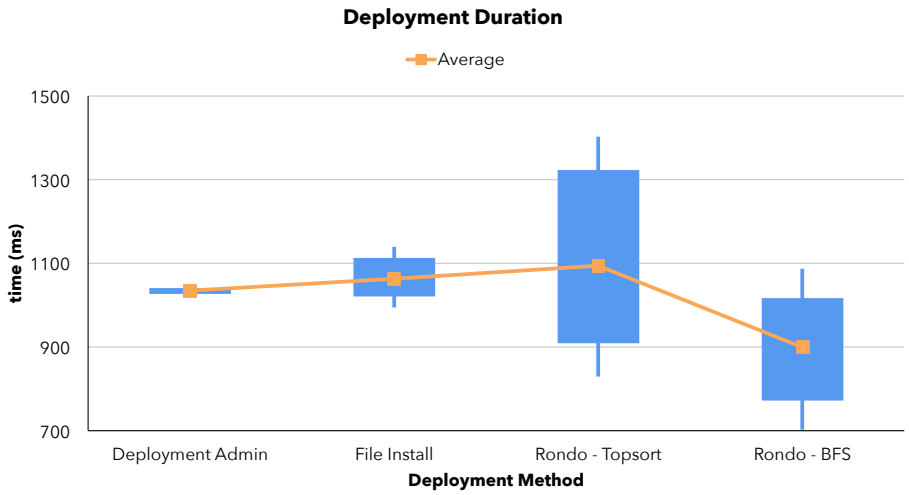


Figure 7.3: Deployment Execution Time Distributions

In case of Rondo deployment process, the dependencies are already declared inside the deployment descriptor and a deployment plan is prepared by the resolver. The Topsort resolver produces sequential deployment plans, in which only one deployment action is executed at a time. This explains the longer deployment duration of Rondo with Topsort resolver. However, Rondo using the BFS resolver produces faster deployments because, the resolver produces deployment plans that contain parallel deployment actions. In turn Rondo executes several deployment actions at the same time. As the deployment plan already takes into account the dependencies between resources, the dependency resolution of bundles conducted by OSGi framework is much faster.

To conclude this section it is useful to compare the development effort for creating descriptors of each deployment method. The table 7.3 compares the number of development files, the number of deployment files, lines of code and lines of configuration need for describing and deploying the test application with each deployment method.

Table 7.3: Test application development efforts

Deployment Method	# of dev. files	# of dep. files	LOC	LOConf
Deployment Admin Package	3	1	0	64
Apache Felix File Install	2	7	0	2
Rondo	2	1	112	46

The first thing to notice is that File Install method for deploying applications is merely copying the bundles and configurations in a watched directory. So there is no deployment descriptors only bundles and configurations for creating instances. The Deployment Admin service uses special archive files called ***deployment package*** for packaging the bundles and other artifacts. Deployment packages contain a manifest that lists their content. This manifest is read and interpreted at runtime by the deployment agent. For creating the deployment package for the test application, the same bundles and configurations are used. An Apache Maven project uses a plugin for creating the deployment package. As for Rondo, the deployment descriptor is coded using the Java DSL, compiled and packaged as an OSGi bundle. The lines of codes corresponds to that of the Java descriptor and the lines of configurations are for the Apache Maven project that creates the bundle. A noticeable trait is that even for a small sized application, the Rondo descriptor contains declarations for resources and dependencies; therefore it requires more lines of code.

7.4 Use of Rondo in Various Deployment Scenarios

This section presents several use cases in different projects where Rondo is used for deployment. Rondo is integrated into execution platforms operating in pervasive and web domains. The primary use cases are first to deploy the technical services that constitute the platform and then on top of that, deploy several applications. Here two projects, iCASA and Wisdom Framework, are presented. Additionally, Rondo is tested on deployment of an Internet of Things gateway platform[6] through the BUTLER project [FP7 BUTLER Project 2013].

7.4.1 iCASA Platform

Along with the development of Rondo and Everest frameworks, the work carried out during this thesis and presented in the previous chapters contributed in the implementation

[6] http://open-platforms.eu/library/butler-smart-gateway/

of a project for pervasive computing. The project, called iCASA, provides a development and execution environment of pervasive applications, specialized in home automation. iCASA is composed of two main parts: a simulator for home automation environment and an execution platform for applications.

iCASA execution platform supports the deployment and execution of home automation applications by providing the following:

- Mechanisms for discovering and reifying physical devices as services using RoSe [Bardin 2010].
- Technical services, such as service for task scheduling, recording user preferences and persisting application data.
- Application development model, based on service-oriented components. In practice, OSGi and iPOJO frameworks are the technical basis for iCASA platform, and this will allow us to directly use our proposal.
- Analysis and introspection tools used to manage running applications and detect any deficiencies.
- A Web interface for viewing and administration of the platform and the applications and services that it is composed of.

Figure 7.4: iCASA Home Simulator

The simulation environment, on the other hand, is used to test one or more applications by imitating a realistic pervasive runtime environment. The figure 7.4 shows an example of such a simulated environment that is represented by this user interface. The

home context and devices shown in this figure are simulated for testing the LightFollowMe application. To achieve this, iCASA provides:

- A virtual home automation environment, representing a house or apartment. This environment includes a physics engine to measure certain characteristics of the environment (light, temperature, noise, etc.). This virtual world also allows to represent the people, their actions, their movements, etc.
- A wide variety of devices that may be simulated and displaced placed in the virtual environment as described above. These devices can directly affect the virtual environment by changing its physical characteristics. For example, a simulated lamp which is lit will increase the brightness of the simulated room in which it is placed.
- A Web interface simulation, which represents graphically the simulated environment, the devices that are present and those who inhabit it. This interface allows to interact directly with the simulated environment by adding new devices, moving users or by activating certain features.
- A scripting language for creating simulation scenarios that tests applications on different combination of configurations.

a. Platform Deployment

As expressed above, the first deployment scenario is the deployment of the platform itself, i.e the technical services that constitute the iCASA framework. The tested platform configuration of iCASA framework version 1.2.6-SNAPSHOT is based on OW2 Chameleon Core and constituted of 107 OSGi bundles. The infrastructure assembly coded in Rondo Java DSL defines 526 resources (including bundles, packages and files) and 1209 dependencies.

The table 7.4 presents comparative results of this deployment. Along with the Baseline framework and the Rondo platform configurations, the table compares a platform that contains iCASA framework (iCASA on the table) and a platform of iCASA with Rondo (iCASA + Rondo). The deployment duration indicates the deployment of iCASA platform on top of Rondo using Topsort and BFS resolvers. As in the previous example these durations are the arithmetic mean value of 10 deployment times.

As shown on the table, Rondo on top of the iCASA framework causes a slower platform start-up, due to the number of bundles and component instances Rondo brings to the platform. From the comparison of idle memory consumptions, an apparent result is that the memory overhead of Rondo is increased but not significantly. This increase is mostly due to the size of the system that is represented by the Everest framework. As for the deployment durations, they reveal the difference between parallel and sequential deployment processes.

Table 7.4: iCASA Framework deployment measurements

Platform	Start-up Duration (ms)	Idle Memory (Mb)	Deployment Duration (ms)
Baseline - Chameleon Core	2694	35.87	–
Rondo - Topsort	5876	46.85	20417
Rondo - BFS	5885	45.89	11213
iCASA	7585	81.41	–
iCASA + Rondo	8872	95.16	–

The assembly that describes the iCASA framework is obtained using the Rondo Cloner (see previous chapter 6.2.6) on an already executing iCASA platform. The Rondo Cloner generates a deployment descriptor that can be edited and build into deployable application descriptions.

b. Application Deployment

The second deployment scenario is the deployment of modules and applications on top of the platform. Leveraging the modular architecture enabled by the OSGi, iCASA project disposes several technical service modules that can be included into the framework dynamically at runtime. In addition to these modules, there is a collection of pervasive applications developed using the home context provided by iCASA.

Before the integration of Rondo, iCASA project experimented with different deployment methods for the installing technical service modules and applications. The first and basic deployment method was using Apache Felix File Install for inserting OSGi bundles and configurations into the platform. More recently, the iCASA project integrated a custom deployment process, which extends the default Deployment Admin Package process. Inside this process, every module is packaged as a deployment package – an archive file including the artifacts and a special manifest that lists the content of the package. Using deployment admin package had several consequences on the life cycle of the project:

- ***In development,*** each module has a Apache Maven project that includes configuration artifacts and builds the deployment package by gathering executable artifacts (i.e. included bundles). The development of these modules was particularly challenging. First, the deployment admin packages do not contain any information for specifying relations between included artifacts. This leads to invent custom mechanisms for circumventing this issue. Developed module often logically extended existing modules. To resolve this issue, the development team invested in developing a build process that included the contents of the extended deployment packages inside the new one.

- ***In deployment,*** the default deployment agent of deployment admin package is extended with a manager that extends and oversees the deployment process. Firstly,

the process is extended with a resource processor that handles configuration artifacts. These artifacts are extracted from the archive and treated as in the case for File Install. Secondly, because the deployment description doesn't include any relation between artifacts, often the installed bundles were not resolved and configured at first try. To overcome this issue, the manager re-invoked the deployment process after a predefined timeout, in order to retry the installation of the deployment package.

The utilization of Rondo for deploying modules and application on iCASA was straightforward. All types of resources needed for these modules – bundles, packages, components, instances, configurations, files – are already included in Rondo Core. Then all that is required is to program the deployment descriptions in Rondo DSL.

Here is a selection of technical service modules and applications that are tested for deployment utilizing Rondo:

Zigbee Module: This module regroups necessary APIs, device proxy implementations, communication and discovery mechanisms for integrating Zigbee devices into iCASA framework. It includes *nrjavaserial*[7] library for serial communication through USB port, where the Zigbee radio dongle is plugged.

Philips Hue Module: This module includes *Philips Hue SDK*[8] and the discovery mechanism for importing Philips Hue lamps into iCASA framework as OSGi services.

Jersey Module: This module includes Jersey[9] core and client bundles for importing and exporting RESTful Web Services as OSGi services into iCASA framework.

Gas Detection Alarm Application: This application uses gas sensors present in the home environment (simulated by iCASA) for detecting increased levels of CO_2 concentration in the air. When the concentration threshold is breached it triggers an alarm using the lighting system (lamps, etc.). In addition to that it sends an e-mail report to a designated person. For this it includes a E-mail API and Service.

Light Follow-Me Application: This application uses motion sensors present in the home environment (simulated by iCASA) for detecting presence in rooms. It regulates the lighting system inside rooms by turning on the lights for occupied rooms and turning off the lights when the room is no longer occupied.

Actimetrics Application: This application registers the occupation rate of rooms and sends the gathered data to a remote server using a RESTful Web Service. It includes the Jersey Module for importing the remote Web Service and sending the actimetry data.

[7] Nrjavaserial: https://github.com/NeuronRobotics/nrjavaserial
[8] Philips Hue SDK: http://developers.meethue.com/
[9] Jersey: https://jersey.java.net

The following table (7.5) presents the comparison of development efforts between Deployment Admin package and Rondo. For Deployment Admin, the table indicates the number of artifacts included inside the deployment package and the number of lines of configuration for Apache Maven project. Rondo deployment descriptions of each module and application are developed using Groovy DSL. The table shows the number of lines of Groovy code developed and the number of resources and dependencies of resulting assembly.

Table 7.5: iCASA Module deployment descriptor development efforts

Module	Deployment Admin		Rondo - Groovy		
	# of artifacts	LOConf	# of resources	# of dependency	LOC
Zigbee	7	66	36	58	56
Philips Hue	3	46	33	40	21
Jersey	3	46	54	65	29
Gas Detection	4	66	25	34	30
Light Follow-Me	1	51	16	15	12
Actimetrics	3	61	81	106	39

The development experiments show that developing Rondo deployment descriptors with Groovy DSL is straightforward. Despite the high complexity of dependencies between resources, the lines of code is restrained, thanks to the preprocessing of deployment. The preprocessing extracts the dependencies between bundles, packages and components; including them into the assembly automatically. This lets developers to concentrate on the business-specific dependencies, such as files, instances and services.

Compared to the previous deployment method in iCASA framework, utilizing the Rondo deployment manager for deployment of modules and applications eliminates the disadvantages mentioned above. The deployment process is well-defined and deployment errors are clearly reported to the user. Furthermore, applications enlisted for deployment are managed at runtime and Rondo allows introspecting these. Any errors occured during the deployment process are registered and available for diagnostics.

7.4.2 Wisdom Framework

The deployment capabilities of Rondo are tested inside another project called Wisdom Framework[10]. Wisdom is a framework for developing modular dynamic web applications. It is based on non-blocking I/O (Netty[11]) and an actor system (Akka[12]), limiting thread and CPU usage. Wisdom is built on top of OSGi, to enable modularity, and on Apache Felix iPOJO, in order to handle the dynamism.

[10]Wisdom Framework: http://wisdom-framework.org/
[11]Netty: http://netty.io/
[12]Akka: http://akka.io/

Wisdom integrates two ideas for development and runtime of web applications. Wisdom eases the complicated build process of modern web applications, which involves HTML files, client-side code, Javascript libraries, stylesheets, templates and medias. It proposes a simple build process that eases the development and testing of applications. During the development process, each change triggers a Apache Maven build process, which compiles, packages and deploys the application.

Secondly, Wisdom provides a modular and dynamic runtime, featuring a stack of technical services that simplifies the development and execution of web applications. This modular stack comprises services such as template engine for static content, JSON libraries for exchanging easily JSON payload, bean validation, Web sockets support, dynamic internationalization support and scheduled and asynchronous task support.

a. Platform Deployment

Modularity and dynamism inherent to Wisdom applies to its own architecture as well as to the applications developed on top of it. This makes Wisdom an adequate candidate to test the capabilities of Rondo. Similar to the previous example, the first deployment scenario is to deploy the platform itself, all of its technical services and configurations. The tested platform configuration of Wisdom framework version 0.6.2 is deployed on OW2 Chameleon Core and constituted of 55 OSGi bundles. The infrastructure assembly developed in Rondo Java DSL declares 461 resources (including bundles, packages and files) and 846 dependencies.

The table 7.6 presents the results of this deployment test. The table compares a platform that contains Wisdom framework and a platform of Wisdom and Rondo. The deployment duration indicates the deployment of Wisdom platform on top of Rondo using Topsort and BFS resolvers. As in the previous example these durations are the arithmetic mean value of 10 deployment times.

Table 7.6: Wisdom Framework deployment measurements

Platform	Start-up Duration (ms)	Idle Memory (Mb)	Deployment Duration (ms)
Baseline - Chameleon Core	2694	35.87	–
Rondo - Topsort	5876	46.85	15894
Rondo - BFS	5885	45.89	9861
Wisdom	5149	75.76	–
Wisdom + Rondo	5181	83.72	–

As previous tests, the start-up and memory overheads are proportional to the size of the platform. The deployment durations are also coherent with the previous tests. The deployment plan created by the BFS resolver implementation fares better in terms of the time it takes in comparison with the Topsort resolver.

b. Application Deployment

As for application deployments, Rondo Groovy DSL is used to develop the deployment descriptors of two applications.

Wisdom Monitor: This application provides an application for monitoring the execution platform. In addition to the application bundle, which provides the main web application, it includes bundles for monitoring the JVM and the OSGi platform.

Wisdom Documentation: This application serves a web page of the Wisdom framework documentation.

Again, thanks to the preprocessing of bundle dependencies, the deployment descriptors are easy to develop, and still, the deployment process proceeds as expected. The following table (see table 7.7) presents the development efforts for these applications.

Table 7.7: Wisdom application deployment descriptor development efforts

Module	Rondo - Groovy		
	# of resources	# of dependency	LOC
Wisdom Monitor	123	190	47
Wisdom Documentation	11	10	11

One of the returns of experience during the experiments with Wisdom framework involved error diagnosis. The first attempt to deploy the Wisdom framework resulted with an error due to unresolved dependencies of a bundle. Rondo deployment manager reported the error back as the result of the deployment process was unsuccessful. A closer inspection revealed that the bundle lack indeed proper manifest metadata for declaring its dependencies. Then two options were possible to overcome this issue. In the immediate, Rondo deployment descriptor for Wisdom framework was updated in order to declare necessary dependencies. Then, the error is reported as a development issue and subsequently the bundle is fixed with correct dependency metadata.

To conclude this section it is useful to present a recapitulation of previous experimentations. The following chart in figure 7.5 outlines the impact of having Rondo on the execution platform. The chart compares the idle memory consumption and start-up durations of baseline, iCASA and Wisdom frameworks, against its counterparts with Rondo.

The start-up durations show a steady increase with the size of the framework in question. The only exception for that is the case for Baseline framework with Rondo deployment manager. The mechanism iPOJO uses for starting component instances explains this increase. iPOJO uses multiple threads to start components that are contained in bundles. Default resource processors for Rondo are contained inside the deployer bundle, which are in this case handled inside a same thread. Furthermore, the memory used by Rondo increases constantly but negligibly with the framework size.

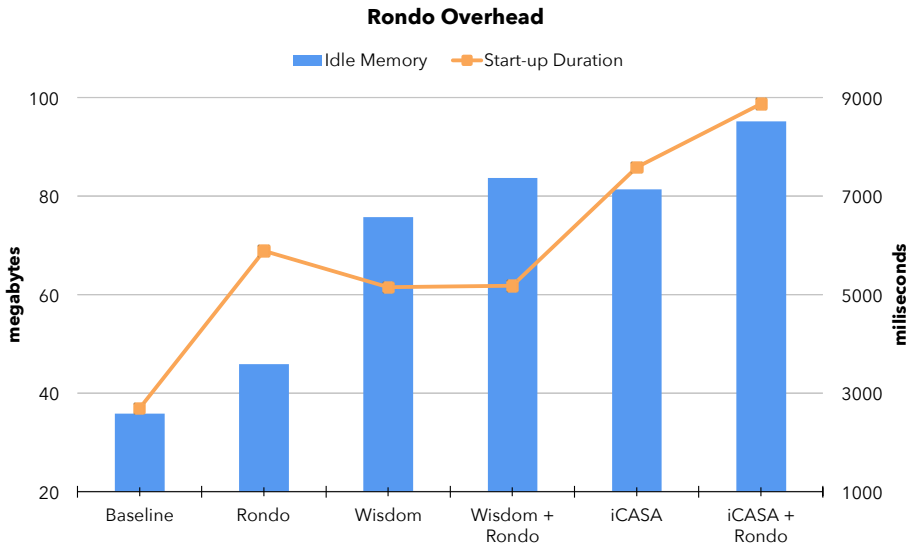


Figure 7.5: Overhead of Rondo

7.5 Dynamic Adaptability in Rondo

In the last part of this section, dynamic adaptation capabilities granted by Rondo are demonstrated using two adaptation scenarios. The first adaptation case involves the adaptation of an application, guided by its deployment description which includes variability. The second case demonstrates a case for updating the technical services that constitute the framework itself.

7.5.1 Application Adaptation

For the first adaptation scenario, consider the test application presented previously in section 7.3.1 of this chapter. The application consisted of a single, non-conditional assembly depicted in the figure 7.3.1. This served to describe the application in a static fashion that did not define any dynamic adaptations. Leveraging its service-oriented modular design and implementation, the application would survive service disruptions or an externally triggered update on its dependencies. However, it would not autonomously change its architecture reacting to the changes.

In order to add self-adaptive capabilities to this application the existing application description is augmented with two conditional assemblies, as shown in the figure 7.6.

Recall that the main non-conditional assembly included one client instance, one server instance and two support instances. The new version of the test application contains the main non-conditional assembly as-is. In addition to that, the first conditional assembly defines a third ***support*** instance, uniquely named `support-3`. Also a dependency between this instance and the ***support*** component description noted c_{sup} is declared.

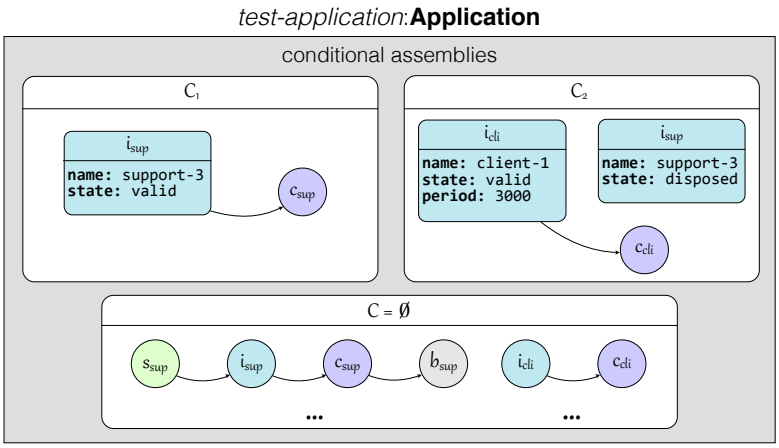


Figure 7.6: Test Application Conditional Assemblies

The second conditional assembly redefines the ***client*** instance, named `client-1`, already contained inside the main assembly, with a new instance configuration. This instance declares its dependency to the ***client*** component description, noted c_{cli}. Furthermore this second assembly redefines the support instance `support-3` with a state **`DISPOSED`**, which is the negative state for instance resource type. This explicit description denotes that the instance `support-3` will not exist inside this assembly.

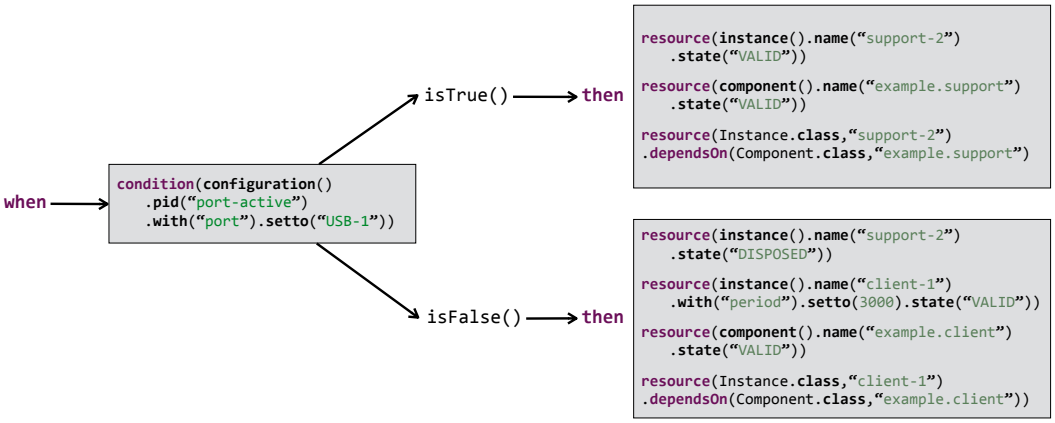


Figure 7.7: Test Application Conditions

As for the conditions of these assemblies, they are defined as complementary and mutually exclusive, i.e. when one is true, the second one is false. The conditions are based on a resource of type configuration (OSGi Configuration Admin configuration). This configuration resource is specified with an id and holds a property value. The schema shown in the figure 7.7 explains the condition cases and assembly descriptions using the portions of the Java DSL code.

As a result of these modifications the test application gained the ability to autonomously adapt to the changes of the configuration. Considering that at the time of

first deployment of the application the configuration resource `active-port`, has the expected value, a third ***support*** instance is included into the application. Then during the course of execution, if the configuration resource does not hold the expected value anymore, due to an internal or external event, the change is detected and a deployment process is triggered to adapt the application. The adapted application includes only two ***support*** instances and a different configuration for ***client*** instance. The inverse case is also valid, according to the state of the condition, the effective application configuration is changed back and forth.

A set of experiments is conducted in order to measure the time cost of these adaptations. The table 7.8 lists the average durations in milliseconds of deployment processes that apply corresponding changes.

Table 7.8: Application adaptation comparison

Action	New Configuration	Configuration Reconf.	New Instance	Instance Reconf.	Application Adaptation
Time(ms)	9	1.16	82	1.43	236

The table compares creation and reconfiguration of iPOJO instances and Configuration Admin configurations and the execution time of the deployment process that adapts the test application. A general remark is that any application adaptation takes longer then the actual changes brought by the deployment actions (new instance, reconfiguration, etc.). This is because the whole application assembly is calculated and validated at the analyze phase, resulting in a longer deployment process. Lastly, note that deployments describing the current state of the platform (for example instance declaration for an already existing instance) take much significantly less time because of idempotence of the deployment actions and process.

7.5.2 Framework Update

The second case that demonstrates the dynamic adaptability capabilities of Rondo is an update scenario within Wisdom framework. Being in active development at the time of writing of this manuscript, the Wisdom framework project made several releases. This section briefly explains the return of experience of using Rondo for deploying and updating Wisdom framework.

With each release of Wisdom framework, the Rondo code that describes the framework deployment is needed to be revised for the new version. This revision task is fairly easy thanks to the use of standard programming language constructs such as fields, parameters and methods. For example, the release version that is shared by all of the Wisdom project artifacts is able to be parameterized into field. This is also valid for bundle dependencies that constitute logical modules that share the same namespace and version. As a result, for the case of updating the Wisdom framework version 0.5.1 to the 0.6.2, the

revision task is in essence the change of a field denoting the framework version in the description source code.

Once the new version of the deployment descriptor is produced, the stake is to push this update to the platforms executing the old version (0.5.1) of the framework. Thanks to the modular dynamic nature of Wisdom framework and applications, the update operation can be applied at runtime. The deployment of new version (0.6.2) conducted by Rondo deployment manager only updated the necessary bundles (14 of 43 bundles forming the Wisdom Framework), leaving the matching resources unchanged. During the deployment process, the update operation of critical technical services disrupted the applications, but once the new version of the framework is up and running, the applications took over. However, only in seldom cases some of the platform services could not handle the dynamism and stopped working.

The last remark about the deployment process is about the kind of utilized deployment plans. In spite of longer deployment durations produced by sequential deployment plans, the framework update tests using the DFS or Topsort resolver implementations fared better in terms of safety of the platform.

7.6 Conclusion

This chapter proposes a validation for the overall contribution of this thesis. Making use of the proposed implementation – Rondo deployment tools – validation cases presented in this chapter proves that the approach adopted in this thesis is pertinent. More specifically, the validation cases evaluate four aspects; the performance, the ability to be used in different deployment scenarios, its advantages for developers and finally the ability for conducting dynamic adaptations.

From the performance acceptability point of view, two separate evaluations are effectuated. The resolver evaluation tested three resolver implementations against various assembly cases and compared their performance. Then in the performance evaluation section, capabilities of Rondo deployment manager is compared against other currently used deployment methods. This section revealed an apparent but acceptable overhead compared to the other methods. This overhead is compensated through improved results in terms of the duration of deployment processes.

The chapter followed by presenting two projects, iCASA and Wisdom frameworks, in which Rondo is tested as the method for deploying the framework and the applications. Such use cases demonstrate the usability and easy adoption process of the approach. The Rondo DSL for programming deployment descriptors plays a central role in this outcome. Along these tests, return of experiences show that the deterministic and fault-tolerant behavior provided by Rondo helps developers for reacting to errors that may occur inside the deployment process.

Finally, the previous section demonstrates dynamic adaptation capabilities granted by Rondo deployment manager. These are presented in two different use cases. The first case illustrates the process of adding variability over an example application, using the Rondo DSL. This simple example of application variability showcases the potential of the proposed deployment process for elaborate continuous adaptation usages. The second use case presents the experience of a framework update operation. This exemplifies how the deployment process handles a large-scale runtime update task.

Chapter 8

Conclusion

" Je ne sais pas ce qui m'attend ni ce qui viendra après tout ceci. "

— Albert Camus, *La peste*

Contents

8.1 Introduction

This thesis studied the continuous deployment in dynamic environments and presented in detail the contributions of this work. The presented contributions comprise the definition of a continuous deployment process, the reference architecture of the deployment manager implementing this process, and a domain-specific language to describe deployments. This work also contributed to the development of a set of deployment tools, called Rondo, which served to highlight and validate the points made throughout this thesis. Rondo is a fully operational prototype and available as an open source project. Capabilities of Rondo deployment manager are tested within different deployment scenarios, using various software projects. This chapter summarizes these propositions and results of this work.

This work gives birth to many research questions and perspectives. The second part details future work possibilities. These involve, first of all, investigating runtime support for applications in dynamic environments and secondly, enhancing existing tool ecosystem to improve testability and ease of use.

8.2 Thesis Summary

This section highlights the contributions of this work. It summarizes the various points raised in this thesis.

8.2.1 Problem Statement

The development of modern applications is a software engineering challenge. It requires providing developers coherent tools and processes to make sure of correct execution and fast software delivery. Dynamism is one of the requirements that is increasingly expected from modern applications. Pervasive environments, for instance, require applications to dynamically evolve at runtime in order for them to blend seamlessly into real environments. Adaptations are necessary to add new functionality to an application, but also to improve quality or to adjust to a new execution context. However, the development of such applications is complex and error-prone. Developers are usually obliged to sacrifice software consistency and dependability in the expense of achieving dynamism.

Furthermore, recent years have witnessed the proliferation of application platforms. This Platform-Application view creates a separation over the type and the control level of software management. Platform providers want to make sure that their platform is working as expected. This requires exhaustive testing of those platforms, against many scenarios and thus exert relatively slow but confident evolution over the software. Conversely, application developers require attracting users with new functionalities as fast as possible. So the applications need evolving more rapidly and dynamically to the changes.

The software development life cycle of applications running in dynamic environments is hindered by the lack of tools that help delivering software rapidly and automatically into environments used for development, testing and production. Traditional approaches fail to address the deployment challenges of dynamic systems.

Emerging practices of continuous deployment is a promising candidate for responding to the deployment needs of dynamic environments. It is based on a set of practices aiming to provide a process for deploying software rapidly and predictably. The continuous deployment for dynamic environments would need to respond to different requirements of execution platforms and applications, updates of separate modules, as well as their reconfigurations to cope with the evolving context.

8.2.2 Contributions

The main objective of this thesis is to enable continuous deployment on dynamic execution environments. The requirement analysis presented as part of this work showed that existing works in this domain are inadequate. In addition to satisfying these requirements, contributions of this work lean specifically on four points. These objectives address the research challenges that are addressed in this thesis (see table 8.1).

Table 8.1: Research challenges and contributed objectives

Objectives	**Challenges**
Reproducibility	*Scalability*, *Heterogeneity*
Fault-tolerance	Distribution, *Industrialization*
Continuous Adaptation	Dynamism, *Context-awareness*
Tooling	Automatization, Testability

The first point is the ***reproducibility*** of deployed software systems. The deployment process must ensure that repeating the same deployment operation in different deployment sites produces the same result on every site. A reproducible deployment process is necessary for large-scale deployment of software systems.

The second point consists of the ***fault-tolerance*** of the deployment process. The deployment includes a series of critical actions that are error-prone. Providing industrial-scale, distributed solutions for deployment depends on the ability of the deployment process to be fault-tolerant.

The third point involves the support of ***continuous adaptation*** of deployed software. The successful deployment of a software system in dynamic environments is not the end of the process, but the beginning of the runtime management of the deployed system. Handling dynamism requires continuous runtime management and adaptation of software.

The last point is the ***tool ecosystem*** that facilitates the work of developers who specify deployments. The specification of the deployment descriptors is a crucial but laborious task. Tools help developers to create and test the deployments they specify.

In the light of these requirements, this work contributes to the specification and the development of a set of deployment facicilities. These facilities comprise:

1. the process definition that allows continuous deployments,
2. the reference architecture for a deployment manager and
3. the domain-specific language for describing deployment tasks.

Central to the proposed deployment facilities is a formalization framework. This framework models the expected state of applications (or any other software to be deployed) and the current state of the deployment platform. These models are based on the generic concept of resource. A resource describes the state of any kind of entity. Using the graph theory, this framework formally describes the deployment process. The formalization framework includes the description of the deployment process, based on the graph theory. This ensures the idempotence of the deployment process, which is crucial for achieving fault-tolerance and reproducibility.

The contributions of this thesis include the reference architecture for the deployment manager that implements the deployment process. The deployment manager specifies the transactional deployment executor which coordinates deployment process. An analysis component is in charge of the runtime management of applications. It enables continuous adaptations by maintaining a monitoring infrastructure and triggering deployment requests when needed.

Conforming to the formalization framework, adaptation requirements of applications are described as variabilities. Variability descriptions are taken into account at application runtime for adapting the application according to the context changes. If a context change triggers an adaptation, the deployment process checks the validity of the application and proceeds with its execution on the platform.

Finally the deployment facilities include a DSL for describing deployments. It constitutes a basis for the tool support and ensures the ease of use of the proposed deployment facility.

The contributions of this thesis are fully developed inside Rondo project. Rondo is a tool suite containing the implementations of the formalization model, the deployment manager and the deployment descriptor DSL. This work also presented the implementation details of Rondo tools. These tools are fully operational and available as open source at `https://github.com/AdeleResearchGroup/Rondo`. In addition to serving for validating the approach and the contributions of this thesis, Rondo is tested against deployment scenarios defined within industrial and research projects.

8.3 Future Work

This work proposed an approach for enabling continuous deployment in dynamic execution environments. Nevertheless, there are many open research questions on the deployment solutions in dynamic environments and much to do to improve the contributions of this thesis. This section looks into some of the perspectives that are revealed over this study.

8.3.1 Improving Support for Applications

This thesis proposes an application description based on the deployment point-of-view. Nevertheless, the notion of application in dynamic, open execution environments leaves a lot more to investigate. The section 5.4.5 on application compatibility discusses the basic conditions for multiple applications to cohabit on a platform. While this ensures the coherence of application resources, no restriction is enforced for applications sharing, accessing and using resources at runtime.

The resource-based application description proposed in this thesis is a good starting point for describing the boundaries and access rights of each application. For example, in Android OS access right to common platform services is based on declarations that come with the application description. Users who authorize the installation of the application, approve the access request. In dynamic execution environments, however, access rights can be provisional and change according to the context. The description of provisional access rights might seem trivial, but the enforcement of these should investigate advanced security mechanims.

Furthermore, sharing application resources is another concern faced by execution platforms. Recall that applications as defined in this work do not possess any belongings towards resources. In this setting certainly applications can enforce constraints on the evolution of resources (using post conditions) but this does not mean that a resource *belongs* to one particular application. The common practice for cohabiting multiple applications is to isolate them in sandboxes. This highly restricts sharing of application resources, although applications still have access to common platform resources. Exploring isolation mechanisms for dynamic open execution environments would contribute to resolve some of the problems encountered in this thesis, such as application undeployment.

8.3.2 Mechanisms for Analyzing and Testing Deployments

The deployment process described in this thesis proposes simple analysis at runtime for transforming applications into deployable entities, that is to *assemblies*. A major phase inside this analysis consists of deciding which application fragments, *conditional assemblies*, are to be deployed. The analysis on different variabilities of the application is the

heart of the continuous adaptability. As expressed earlier in this document, future work is bound to investigate the possibilities for this decision function.

Similar research questions are already explored in the context of SPL feature models. A major question is what knowledge is needed by this decision function to be able to choose between variabilities. This work paves the way for more advanced adaptation scenarios by associating a well-defined deployment process with self-adaptation possibilities. Autonomic Computing solutions can be employed to enhance the self-adaptable applications with self-optimization, self-healing and self-protection functionalities.

The continuous adaptability implies analyzing application variabilities at runtime. However, deployment descriptors can also be analyzed beforehand for testing purposes. Using combinatorial testing methods application descriptions can be exposed against different platform configurations and state changes. This would allow testing the adaptation logic contained inside variabilities.

The deployment process proposed in this thesis involves analyzing dependencies between resources, according to the deployment description, in order to calculate a deployment plan. Therefore all dependencies between resources are expected to be explicitly included inside the deployment description. Even though this is needed for guaranteeing the determinism of the deployment process, specifying all dependencies can be laborious and it can hinder the development.

As argumented earlier, this process does not include a phase for resolving dependencies by soliciting an artifact repository. Nevertheless, as discussed in the section 5.4.6, such a mechanism can be integrated inside an development environment (IDE) that help completing deployment descriptors with dependency resolution information.

8.3.3 Distributed Continuous Deployment

This work deliberately excluded the problem of deploying software to multiple target platforms in distributed environments. Next logical step is looking for expanding this work into distributed environments. Basically put, the distributed deployment is an orchestration of deployments in multiple target machines. As a matter of fact, many properties ensured by this work, such as fault-tolerance, determinism and introspection are crucial for distributed deployment.

Some of the works in this direction are already presented in section 3.5.5 of this document. In order to provide continuous deployment in distributed environments, the planning function should be revised for taking into account spatial and temporal constraints over target machines.

With the emergence of many Cloud computing providers, the possibilities for distributed deployment are increasing. Most of these providers offer APIs so that deployment automation tools can manipulate Cloud resources as VMs, computing infrastructures or

platforms. This unlocks new possibilities for easily managing distributed deployment. There are many tools that already help developers and system administrators in this direction. Docker (https://www.docker.com/) provides a lightweight runtime based on Linux containers. It offers a set of tools for packaging applications and automating testing and deployment workflows. Roboconf (http://roboconf.net/) is an automation tool for coordinating deployment to multiple Cloud targets.

Extending the approach proposed in this thesis with these capabilities would enable dynamically adaptive deployment in distributed environments.

8.3.4 Integration into Deployment Pipeline

This work proposes a deployment solution capable of being used in continuous deployment scenarios. However, it does not provide the complete deployment pipeline. Integrating the contributions of this thesis into an end-to-end software development process would unlock the full potential of the promise of continuous deployment.

The deployment descriptor language provided in this work would play a pivotal role inside such a deployment pipeline. Each commit during the development would trigger a set of tests before being ready to be deployed into production environments. Differently to the current testing practices, to enable the triggered updates, a testing platform, having the same current state of production environments, would receive the update, in order to test the update behavior and possible errors.

In the pervasive computing scenarios, platforms like iCASA that simulate physical environments, can be used in the context of testing dynamic behavior of applications. These tests can be automatized in part of the deployment pipeline.

Appendix A

Proof of Assembly Join Associativity

Proof. Associativity property can be proven by induction. Consider a transition function of any non-empty assembly X, $\mathcal{T}_X = \delta$ such that $\delta(A@B) = \delta(A)@B$. Indeed this is valid for every non-empty assembly X, as shown in following A.1.

$$\begin{aligned} A@B &= \mathcal{T}_A(B) \\ \mathcal{T}_X(A@B) &= \mathcal{T}_X(\mathcal{T}_A(B) \\ &= \mathcal{T}_X \circ \mathcal{T}_A(B) \\ &\Longrightarrow \\ \exists Y \in \mathbb{A}^*, \mid Y &= \mathcal{T}_X(A) \\ Y &= \mathcal{T}_X(\mathcal{T}_A(\varepsilon)) \\ \mathcal{T}_Y(\varepsilon) &= \mathcal{T}_X \circ \mathcal{T}_A(\varepsilon) \\ \mathcal{T}_Y &= \mathcal{T}_X \circ \mathcal{T}_A \\ \mathcal{T}_X \circ \mathcal{T}_A(B) &= \mathcal{T}_Y(B) = Y@B \\ &= \mathcal{T}_X(A)@B \end{aligned} \tag{A.1}$$

Then using the identity element and this transition δ as the successor function, the associativity can be proven by applying induction.

$$A, B, C \in \mathbb{A}^*, (A@B)@C \stackrel{?}{=} A@(B@C)$$

For the base case consider $A = \varepsilon$,

$$\begin{aligned} (\varepsilon@B)@C &\stackrel{?}{=} \varepsilon@(B@C) \\ (\varepsilon@B)@C &= (B)@C = B@C \\ &= B@C = \varepsilon@(B@C) \end{aligned}$$

For the induction case consider that the hypothesis is true, assuming that for an assembly A, $A@(B@C) = (A@B)@C$. Then,

$$\begin{aligned} \delta(A)@(B@C) &\stackrel{?}{=} (\delta(A)@B)@C \\ \delta(A)@(B@C) &= \delta(A@(B@C)) \\ &= \delta((A@B)@C) \\ &= \delta(A@B)@C \\ &= (\delta(A)@B)@C \end{aligned} \tag{A.2}$$

Then the induction case is true for $\delta(A)$. Therefore the associativity property $(A@B)@C = A@(B@C)$ is true for all assemblies. □

Appendix B

Description Language Grammar

```
Property ::= StringLiteral ':' StringLiteral
Properties ::= Property*
Id ::= IdLiteral
Type ::= IdLiteral
Name ::= 'name' ':' StringLiteral
State ::= 'state' ':' StringLiteral
Version ::= 'version' ':' VersionLiteral
Repository ::= '{' Name ',' 'url' ':' UrlLiteral '}'
Repositories ::= '[' Repository (',' Repository )*  ']'
GenericResource ::= '{' Type (',' Name)?
      (',' State)? ',' Properties '}'
ResourceDeclaration ::= Id GenericResource? 'dependsOn' Id (',' Id)*
Assembly ::= 'resource' ResourceDeclaration
                  ('\\' 'resource' ResourceDeclaration)*
Fact ::= ('true' | 'false')
Condition ::= '{' GenericResource ':' Fact '}'
Conditions ::= '[' Condition (',' Condition)*  ']'
ConditionalAssembly ::= (
      ('when' ':' Conditions 'then' '{' Assembly '}' )
      | ('with' '{' Assembly '}') )
Application ::= Id ',' Name ','
```

Listing B.1: EBNF Grammar of the description language

Appendix C

Publications

- Escoffier, Clement; Gunalp Ozan; Lalanda, Philippe, “Requirements to Pervasive System Continuous Deployment”, International Workshop on Self-Managing Pervasive Service Systems (SeMaPS), 2013
- Gurgen, Levent; Gunalp, Ozan; Benazzouz, Yazid; Gallissot, Mathieu, "Self-aware cyber-physical systems and applications in smart buildings and cities," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013
- Gunalp, Ozan; Gurgen, Levent; Lestideau, Vincent; Lalanda Philippe, “Autonomic Pervasive Applications Driven by Abstract Specifications”, International workshop on Self-aware internet of things (Self-IoT), 2012

List of Figures

List of Tables

List of Algorithms

Bibliography

[Andersson 2000] J. Andersson. *A deployment system for pervasive computing.* In Software Maintenance, 2000. Proceedings. International Conference on, pages 262–270, 2000. (page 107.)

[Arshad 2001] Naveed Arshad, Dennis Heimbigner et Alexander L Wolf. *Deployment and dynamic reconfiguration planning for distributed software systems.* pages 39–46, July 2001. (pages 108 and 109.)

[Atzori 2010] Luigi Atzori, Antonio Iera et Giacomo Morabito. *The Internet of Things: A survey.* Computer Networks, vol. 54, no. 15, pages 2787–2805, October 2010. (page 13.)

[Bailey 1997] Ed Bailey. Maximum RPM. Red Hat Software Inc., February 1997. (pages 70 and 106.)

[Baldauf 2007] Matthias Baldauf, Schahram Dustdar et Florian Rosenberg. *A survey on context-aware systems.* International Journal of Ad Hoc and Ubiquitous Computing, vol. 2, no. 4, pages 263–277, June 2007. (page 14.)

[Banavar 2000] Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy Sussman et Deborra Zukowski. *Challenges: An Application Model for Pervasive Computing.* In Proceedings of the 6th Annual International Conference on Mobile Computing and Networking, MobiCom '00, pages 266–274, New York, NY, USA, 2000. ACM. (page 33.)

[Bardin 2010] Jonathan Bardin, Clément Escoffier et Philippe Lalanda. *Towards an Automatic Integration of Heterogeneous Services and Devices.* In Proceedings of the Services Computing Conference (APSCC), pages 171–178. IEEE Computer Society, December 2010. (page 210.)

[Baresi 2010] L Baresi et C Ghezzi. *The disappearing boundary between development-time and run-time.* Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010. (pages 4 and 60.)

[Bauer 2002] Martin Bauer, Christian Becker et Kurt Rothermel. *Location models from the perspective of context-aware applications and mobile ad hoc networks.* Personal and Ubiquitous Computing, vol. 6, no. 5, pages 322–328, 2002. (page 14.)

[Becker 2003] Christian Becker, Gregor Schiele, Holger Gubbels et Kurt Rothermel. *Base-a micro-broker-based middleware for pervasive computing.* pages 443–451, 2003. (pages 35 and 107.)

[Becker 2004] Christian Becker, Marcus Handte, Gregor Schiele et Kurt Rothermel. *Pcom-a component system for pervasive computing*. pages 67–76, 2004. (pages 35, 107 and 108.)

[Belguidoum 2008] Meriem Belguidoum et Fabien Dagnat. *Formalization of Component Substitutability*. Electronic Notes in Theoretical Computer Science, vol. 215, pages 75–92, June 2008. (page 169.)

[Bencomo 2010] Nelly Bencomo, Jaejoon Lee et Svein Hallsteinsen. *How dynamic is your Dynamic Software Product Line?* 2010. (page 62.)

[Bernstein 1996] Philip A. Bernstein. *Middleware: A Model for Distributed System Services*. Commun. ACM, vol. 39, no. 2, pages 86–98, February 1996. (page 29.)

[Bertolino 2009] Antonia Bertolino, Guglielmo Angelis, Lars Frantzen et Andrea Polini. *The PLASTIC Framework and Tools for Testing Service-Oriented Applications*. Software Engineering, January 2009. (page 35.)

[Boehm 2003] B. Boehm et R Turner. Balancing Agility and Discipline: A Guide for the Perplexed. Pearson Education, 2003. (page 45.)

[Bohn 2005] Jürgen Bohn, Vlad Coroamă, Marc Langheinrich, Friedemann Mattern et Michael Rohs. *Social, economic, and ethical implications of ambient intelligence and ubiquitous computing*. In Ambient intelligence, pages 5–29. Springer, 2005. (page 23.)

[Bosch 2000] Jan Bosch. Design and use of software architectures: adopting and evolving a product-line approach. Pearson Education, 2000. (page 61.)

[Bosch 2002] Jan Bosch. *Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization*. In GaryJ. Chastek, editeur, Software Product Lines, volume 2379 of *Lecture Notes in Computer Science*, pages 257–271. Springer Berlin Heidelberg, 2002. (page 105.)

[Bouchenak 2006] S Bouchenak, N De Palma, D Hagimont et C Taton. *Autonomic management of clustered applications*. 2006 IEEE International Conference on Cluster Computing, pages 1–11, 2006. (page 78.)

[Broadband Forum 2013] Broadband Forum. *TR-069 CPE WAN Management Protocol (CWMP)*. Technical report, 2013. (page 35.)

[Bruneton 2004] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma et Jean-Bernard Stefani. *An open component model and its support in Java*. In Component-Based Software Engineering, pages 7–22. Springer, 2004. (page 35.)

[Buckley 2005] Alex Buckley. *A Model of Dynamic Binding in .NET*. In Alan Dearle et Susan Eisenbach, editeurs, Component Deployment, volume 3798 of *Lecture Notes in Computer Science*, pages 149–163. Springer Berlin Heidelberg, 2005. (page 168.)

[Bures 2006] T. Bures, P. Hnetynka et F. Plasil. *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model.* In Fourth International Conference on Software Engineering Research, Management and Applications, 2006., pages 40–48, 2006. (page 107.)

[Carzaniga 1997] Antonio Carzaniga. *A Characterization of the Software Deployment Process and a Survey of related Technologies.* 1997. (pages 49, 50, 51 and 54.)

[Cassou 2010] Damien Cassou, Julien Bruneau et Charles Consel. *A tool suite to prototype pervasive computing applications.* In 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010, pages 820–822, 2010. (page 35.)

[Cetina 2008] Carlos Cetina, Vicente Pelechano, Pablo Trinidad et Antonio Ruiz Cortés. *An Architectural Discussion on DSPL.* In Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops), pages 59–68, 2008. (pages 62 and 105.)

[Church 1936] Alonzo Church et J Barkley Rosser. *Some properties of conversion.* Transactions of the American Mathematical Society, vol. 39, no. 3, pages 472–482, 1936. (page 133.)

[Coghlan 2005] Brian Coghlan, John Walsh, Geoff Quigley, David O'Callaghan, Stephen Childs et Eamonn Kenny. *Principles of Transactional Grid Deployment.* In Advances in Grid Computing - EGC 2005, volume 3470 of *Lecture Notes in Computer Science*, pages 88–97. Springer Berlin Heidelberg, 2005. (page 102.)

[Conway 1968] Melvin E. Conway. *How Do Committees Invent?* Datamation, April 1968. (page 42.)

[Coutaz 2005] Joëlle Coutaz, James L Crowley, Simon Dobson et David Garlan. *Context is key.* Communications of the ACM, vol. 48, no. 3, pages 49–53, 2005. (page 151.)

[Dearle 2004] A. Dearle, G. N C Kirby et A.J. McCarthy. *A framework for constraint-based development and autonomic management of distributed applications.* In Autonomic Computing, 2004. Proceedings. International Conference on, pages 300–301, 2004. (pages 77, 108 and 109.)

[Dearle 2007] Alan Dearle. *Software deployment, past, present and future.* In 2007 Future of Software Engineering, pages 269–284. IEEE Computer Society, 2007. (pages 52, 61 and 149.)

[Detlefs 2004] David Detlefs, Christine Flood, Steve Heller et Tony Printezis. *Garbage-first Garbage Collection.* In Proceedings of the 4th International Symposium on Memory Management, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM. (page 207.)

[Dey 2001] Anind K Dey. *Understanding and Using Context.* Personal and Ubiquitous Computing, vol. 5, no. 1, pages 4–7, 2001. (pages 15 and 151.)

[Diaconescu 2008] Ada Diaconescu, Johann Bourcier et Clement Escoffier. *Autonomic iPOJO: Towards Self-Managing Middleware for Ubiquitous Systems.* Networking and Communications, 2008. WIMOB '08. IEEE International Conference on Wireless and Mobile Computing,, pages 472–477, 2008. (page 103.)

[Dijkstra 1974] Edsger W. Dijkstra. *Self-stabilizing Systems in Spite of Distributed Control.* Commun. ACM, vol. 17, no. 11, pages 643–644, November 1974. (page 21.)

[Dis 2014] Distributed Management Task Force. *Common Information Model (CIM) Specification*, June 2014. (page 95.)

[Dowling 2001] Jim Dowling et Vinny Cahill. *The K-Component Architecture Meta-Model for Self-Adaptive Software.* In Metalevel Architectures and Separation of Crosscutting Concerns, pages 81–88. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2001. (page 35.)

[Druilhe 2013] Rémi Druilhe. *L'Efficience Énergétique des Services dans les Systèmes Répartis Hétérogènes et Dynamiques : Application à la Maison Numérique.* These, Université des Sciences et Technologie de Lille - Lille I, December 2013. (page 21.)

[Dubus 2007] J Dubus et P Merle. *Applying omg d&c specification and eca rules for autonomous distributed component-based systems.* Models in Software Engineering, pages 242–251, 2007. (page 78.)

[Ehrig 1973] Hartmut Ehrig, Michael Pfender et Hans Jürgen Schneider. *Graph-grammars: An algebraic approach.* In Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on, pages 167–180. IEEE, 1973. (page 128.)

[Ehrig 1979] Hartmut Ehrig. *Introduction to the algebraic theory of graph grammars (a survey).* In Graph-Grammars and Their Application to Computer Science and Biology, pages 1–69. Springer, 1979. (page 130.)

[eMarketer 2014] eMarketer. *Worldwide Mobile Phone Users: H1 2014 Forecast and Comparative Estimates.* Market research, 2014. (page 4.)

[Epstein 1998] Brian Epstein et Eli Zelkha. *Ambient Intelligence.* http://www.epstein.org/brian/ambient_intelligence.htm, 1998. (page 12.)

[Escoffier 2006] Clément Escoffier, Didier Donsez et Richard S Hall. *Developing an OSGi-like service platform for .NET.* IEEE Consumer Communications and Networking Conference (CCNC'06), 2006. (page 169.)

[Escoffier 2008] Clement Escoffier. *iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques*. These, Université Joseph-Fourier - Grenoble I, December 2008. (pages 35 and 112.)

[Escoffier 2013a] Clément Escoffier, Pierre Bourret et Philippe Lalanda. *Managing Dynamism in Service Dependencies*. In IEEE International Conference on Services Computing, Los Alamitos, CA, USA, June 2013. IEEE Computer Society. (page 177.)

[Escoffier 2013b] Clement Escoffier, Ozan Günalp et Philippe Lalanda. *Requirements to Pervasive System Continuous Deployment*. The 2nd International Workshop on Self-Managing Pervasive Service Systems (SeMaPS 2013), 2013. (pages 98 and 151.)

[Evans 2007] Dave Evans. *The Internet of Things: How the Next Evolution of the Internet Is Changing Everything*. april 2007. (page 5.)

[Fabry 1976] R S Fabry. *How to design a system in which modules can be changed on the fly*. In ICSE '76: Proceedings of the 2nd international conference on Software engineering. IEEE Computer Society Press, October 1976. (page 112.)

[Fielding 2000] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. (page 152.)

[Flissi 2008] Areski Flissi, J Dubus, Nicolas Dolet et Philippe Merle. *Deploying on the Grid with DeployWare*. In 2008 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid), pages 177–184. IEEE, October 2008. (page 78.)

[Floch 2006] J Floch, S Hallsteinsen, E Stav, F Eliassen, K Lund, E Gjorven, S ICT et N Trondheim. *Using architecture models for runtime adaptability*. IEEE software, vol. 23, no. 2, pages 62–70, 2006. (page 22.)

[Fouquet 2012] François Fouquet, Erwan Daubert, Noël Plouzeau, Olivier Barais, Johann Bourcier, Arnaud Blouin*et al.* *Kevoree: une approche model@ runtime pour les systèmes ubiquitaires*. In UbiMob2012, 2012. (page 35.)

[FP7 BUTLER Project 2013] FP7 BUTLER Project. *uBiquitous, secUre inTernet-of-things with Location and contExt-awareness*. Delivrable, 2013. (pages 19, 209 and 237.)

[Garey 1979] Michael R. Garey et David S. Johnson. Computers and intractability; a guide to the theory of np-completeness. W. H. Freeman & Co., New York, NY, USA, 1979. (page 59.)

[Garlan 2002] David Garlan, Dan Siewiorek, Asim Smailagic et Peter Steenkiste. *Project Aura: Toward Distraction-Free Pervasive Computing*. IEEE Pervasive Computing, vol. 1, no. 2, April 2002. (page 34.)

[Goldsack 2003] Patrick Goldsack, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray et Peter Toft. *SmartFrog: Configuration and Automatic Ignition of Distributed Applications*. In In: HP Openview University Association Conference (HP OVUA, pages 1–9, 2003. (page 76.)

[Greenfield 2006] Adam Greenfield. Everyware : the dawning age of ubiquitous computing. New Riders, Berkeley, CA, 2006. (page 12.)

[Grimm 2003] R Grimm. *System support for pervasive applications*. Future directions in distributed computing, 2003. (page 13.)

[Hall 1999] Richard S. Hall, Dennis Heimbigner et Alexander L. Wolf. *A Cooperative Approach to Support Software Deployment Using the Software Dock*. In Proceedings of the 21st International Conference on Software Engineering, ICSE '99, pages 174–183, New York, NY, USA, 1999. ACM. (pages 49, 77, 108 and 109.)

[Hallsteinsen 2008] S. Hallsteinsen, M. Hinchey, Sooyong Park et K. Schmid. *Dynamic Software Product Lines*. Computer, vol. 41, no. 4, pages 93–95, 2008. (page 62.)

[Hallsteinsen 2012] S Hallsteinsen, K Geihs, N Paspallis, F Eliassen, G Horn, J Lorenzo, A Mamelli et G A Papadopoulos. *A development framework and methodology for self-adapting applications in ubiquitous computing environments*. Journal of Systems and Software, vol. 85, no. 12, pages 2840–2859, December 2012. (page 30.)

[Hansmann 2003] Uwe Hansmann. Pervasive computing : the mobile world. Springer, Berlin New York, 2003. (page 12.)

[Hawley 1997] Michael Hawley, R Dunbar Poor et Manish Tuteja. *Things that think*. Personal Technologies, vol. 1, no. 1, pages 13–20, 1997. (page 12.)

[Heimbigner 1998] Dennis Heimbigner, Andre Van der Hoek, Richard S Hall, Alexander L Wolf, Antonio Carzaniga et Alfonso Fuggetta. *A Characterization Framework for Software Deployment Technologies*. 1998. (pages 65, 68 and 98.)

[Helal 2005] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura et E. Jansen. *The Gator Tech Smart House: a programmable pervasive space*. Computer, vol. 38, no. 3, pages 50–60, 2005. (pages 17, 24 and 237.)

[Hitcents 2014] Hitcents. *Hanx Writer*. `http://www.hitcents.com/b2b/work/hanx`, 2014. Accessed: 2014-09-01. (page 4.)

[Hoareau 2008] Didier Hoareau et Yves Mahéo. *Middleware support for the deployment of ubiquitous software components*. Personal and Ubiquitous Computing, vol. 12, no. 2, pages 167–178, 2008. (pages 107, 108 and 109.)

[Horn 2001] P Horn. Autonomic computing: IBM's perspective on the State of Information Technology, October 2001. (page 95.)

[Huebscher 2004] MC Huebscher et JA McCann. *Adaptive middleware for context-aware applications in smart-homes.* Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing, pages 111–116, 2004. (page 30.)

[Humble 2010] Jez Humble et David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional, 1st édition, 2010. (pages 89, 90 and 237.)

[Humble 2011] Jez Humble et Joanne Molesky. *Why Enterprises Must Adopt Devops to Enable Continuous Delivery.* Cutter IT Journal, vol. 24, no. 8, pages 6–12, August 2011. (page 89.)

[Hunkeler 2008] U. Hunkeler, Hong Linh Truong et A Stanford-Clark. *MQTT-S – A publish/subscribe protocol for Wireless Sensor Networks.* In Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on, pages 791–798, Jan 2008. (page 21.)

[IEE 2008] *ISO/IEC/IEEE Standard for Systems and Software Engineering - Software Life Cycle Processes.* IEEE STD 12207-2008, pages c1–138, 2008. (page 40.)

[Kephart 2003] Jeffrey O. Kephart et David M. Chess. *The Vision of Autonomic Computing.* Computer, vol. 36, no. 1, pages 41–50, January 2003. (pages 63, 96, 103 and 150.)

[Kidd 1999] Cory D Kidd, Robert Orr, Gregory D Abowd, Christopher G Atkeson, Irfan A Essa, Blair MacIntyre, Elizabeth Mynatt, Thad E Starner et Wendy Newstetter. *The aware home: A living laboratory for ubiquitous computing research.* In Cooperative buildings. Integrating information, organizations, and architecture, pages 191–198. Springer, 1999. (page 16.)

[King 2006] Jeffrey King, Raja Bose, Hen-I Yang, Steven Pickles et Abdelsalam Helal. *Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces.* In Local Computer Networks, Proceedings 2006 31st IEEE Conference on, pages 630–638. IEEE, 2006. (pages 24 and 35.)

[Krakowiak 2007] Sacha Krakowiak. *Middleware Architecture with Patterns and Frameworks*, 2007. (pages 28, 34 and 160.)

[Kramer 2007] Jeff Kramer et Jeff Magee. *Self-managed systems: an architectural challenge.* In Future of Software Engineering, 2007. FOSE'07, pages 259–268. IEEE, 2007. (page 99.)

[Kvarnström 2001] Jonas Kvarnström et Patrick Doherty. *TALplanner: A Temporal Logic Based Forward Chaining Planner.* Annals of Mathematics and Artificial Intelligence, vol. 30, no. 1-4, pages 119–169, Mars 2001. (page 103.)

[Lalanda 2014] Philippe Lalanda, Catherine Hamon et Clément Escoffier. *Cilia: An autonomic service bus for pervasive environments.* Proceedings of the 11th IEEE International Conference on Services Computing (SCC), 2014. (pages 18 and 237.)

[Laprie 1992] J C Laprie, A Avizienis et H Kopetz. *Dependability: Basic Concepts and Terminology.* International Federation for Information Processing WG 10.4 on Dependable Computing and Fault Tolerance, February 1992. (page 4.)

[Larman 2003] Craig Larman et Victor R Basili. *Iterative and incremental developments. a brief history.* Computer, vol. 36, no. 6, pages 47–56, 2003. (page 45.)

[Lehman 1980] M M Lehman. *Programs, life cycles, and laws of software evolution.* In Proceedings of the IEEE, pages 1060–1076, 1980. (page 47.)

[Lim 2012] Jong Hyun Lim, Andong Zhan, Evan Goldschmidt, JeongGil Ko, Marcus Chang et Andreas Terzis. *HealthOS: a platform for pervasive health applications.* In Proceedings of the Second ACM Workshop on Mobile Systems, Applications, and Services for HealthCare, mHealthSys '12, pages 41–46, New York, NY, USA, 2012. ACM. (page 35.)

[Liskov 1994] Barbara H. Liskov et Jeannette M. Wing. *A Behavioral Notion of Subtyping.* ACM Trans. Program. Lang. Syst., vol. 16, no. 6, pages 1811–1841, November 1994. (page 104.)

[Liu 2006] Yu David Liu et Scott F. Smith. *A Formal Framework for Component Deployment.* In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, pages 325–344, New York, NY, USA, 2006. ACM. (pages 52 and 169.)

[Mattern 2001] F Mattern. *The vision and technical foundations of ubiquitous computing.* Upgrade European Online Magazine, pages 5–8, 2001. (page 13.)

[Mattern 2005] Friedemann Mattern. Ubiquitous Computing: Scenarios for an informatized world, pages 145–163. Springer-Verlag, 2005. (page 12.)

[Maurel 2010] Y Maurel, Ada Diaconescu et Philippe Lalanda. *CEYLON: A Service-Oriented Framework for Building Autonomic Managers.* 2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems, pages 3–11, 2010. (page 103.)

[McKinley 2004] P.K. McKinley, S.M. Sadjadi, E.P. Kasten et B.H.C. Cheng. *Composing adaptive software.* Computer, vol. 37, no. 7, pages 56–64, 2004. (page 98.)

[Medvidovic 1996] Nenad Medvidovic. *ADLs and dynamic architecture changes.* In Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops, pages 24–27. ACM, 1996. (page 61.)

[Medvidovic 2000] Nenad Medvidovic et Richard N. Taylor. *A classification and comparison framework for software architecture description languages.* IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 70–93, 2000. (page 60.)

[Medvidovic 2007] Nenad Medvidovic et Sam Malek. *Software deployment architecture and quality-of-service in pervasive environments.* In International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting, pages 47–51, New York, NY, USA, 2007. ACM. (page 103.)

[Microsoft 1998] Microsoft. *Windows Management Instrumentation (WMI).* http://msdn.microsoft.com/en-us/library/aa394582(v=vs.85).aspx, 1998. (page 95.)

[Mikic-Rakic 2002] Marija Mikic-Rakic et Nenad Medvidovic. *Architecture-level support for software component deployment in resource constrained environments.* In Component Deployment, pages 31–50. Springer, 2002. (page 77.)

[Miller] Brent Miller. *Can you CHOP up autonomic computing?* http://www.ibm.com/developerworks/autonomic/library/ac-edge4/. (page 158.)

[Moore 1965] Gordon E Moore*et al. Cramming more components onto integrated circuits*, 1965. (page 16.)

[Nelson-Smith 2011] Stephen Nelson-Smith. Test-Driven Infrastructure with Chef. O'Reilly, 2011. (page 106.)

[Object Management Group 2006a] Object Management Group. The corba component model specification - version 4.0. OMG, April 2006. (page 73.)

[Object Management Group 2006b] Object Management Group. Deployment & Configuration of Component-based Distributed Applications Specification - version 4.0. OMG, April 2006. (pages 48, 51, 55 and 108.)

[Oreizy 1999] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum et Alexander L Wolf. *An architecture-based approach to self-adaptive software.* Intelligent Systems and Their Applications, IEEE, vol. 14, no. 3, pages 54–62, 1999. (pages 63 and 100.)

[OSGi Alliance 2007] OSGi Alliance. *OSGi Service Platform Release 4.* [Online]. Available: http://www.osgi.org/Main/HomePage., 2007. (pages 74 and 206.)

[Parashar 2006] M Parashar, H Liu, Z Li, V Matossian, C Schmidt, G Zhang et S Hariri. *AutoMate: Enabling Autonomic Applications on the Grid.* Cluster Computing, vol. 9, no. 2, pages 161–174, April 2006. (page 103.)

[Parra 2009] Carlos Parra, Xavier Blanc et Laurence Duchien. *Context awareness for dynamic service-oriented product lines.* In Proceedings of the 13th International Software Product Line Conference, pages 131–140. Carnegie Mellon University, 2009. (page 105.)

[Parrish 2001] Allen Parrish, Brandon Dixon et David Cordes. *A conceptual foundation for component-based software deployment.* Journal of Systems and Software, vol. 57, no. 3, pages 193–200, 2001. (page 168.)

[Peter Mell and Tim Grance 2011] Peter Mell and Tim Grance. *The NIST Definition of Cloud Computing*, 2011. (pages 79 and 80.)

[Poppendieck 2009] M Poppendieck et T Poppendieck. Leading Lean Software Development: Results Are not the Point. Addison-Wesley Signature Series (Beck). Pearson Education, 2009. (page 89.)

[Poppendieck 2012] Mary Poppendieck et Michael A Cusumano. *Lean software development: A tutorial.* Software, IEEE, vol. 29, no. 5, pages 26–32, 2012. (page 87.)

[Ramalingam 2013] Ganesan Ramalingam et Kapil Vaswani. *Fault tolerance via idempotence.* In POPL '13: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM Request Permissions, January 2013. (page 145.)

[Ratner 2003] M.A. Ratner et D. Ratner. Nanotechnology: A gentle introduction to the next big idea. Safari Tech Books Online. Prentice Hall, 2003. (page 21.)

[Román 2002] M. Román, C. Hess, R. Cerqueira, A Ranganathan, R.H. Campbell et K. Nahrstedt. *Gaia: a middleware platform for active spaces.* ACM SIGMOBILE Mobile Computing and Communications Review, vol. 6, no. 4, pages 65–67, 2002. (page 34.)

[Ronzani 2009] D. Ronzani. *The battle of concepts: Ubiquitous Computing, pervasive computing and ambient intelligence in Mass Media.* Ubiquitous Computing and Communication Journal. v4 i2, 2009. (page 13.)

[Rouvoy 2009] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli et Ulrich Scholz. *Software Engineering for Self-Adaptive Systems.* chapitre MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments, pages 164–182. Springer-Verlag, Berlin, Heidelberg, 2009. (page 35.)

[Rudolph 2001] Larry Rudolph. *Project oxygen: Pervasive, human-centric computing–an initial experience.* In Advanced Information Systems Engineering, pages 1–12. Springer, 2001. (page 34.)

[Salehie 2009] Mazeiar Salehie et Ladan Tahvildari. *Self-adaptive software: Landscape and research challenges.* ACM Trans. Auton. Adapt. Syst., vol. 4, no. 2, pages 14:1–14:42, May 2009. (page 63.)

[Satyanarayanan 2001] M Satyanarayanan. *Pervasive computing: vision and challenges.* Personal Communications, IEEE, vol. 8, no. 4, pages 10–17, 2001. (pages 12, 13 and 112.)

[Schiele 2010] Gregor Schiele, Marcus Handte et Christian Becker. *Pervasive Computing Middleware.* In Hideyuki Nakashima, Hamid Aghajan et JuanCarlos Augusto, editeurs, Handbook of Ambient Intelligence and Smart Environments, pages 201–227. Springer US, 2010. (page 22.)

[Schilit 1994] Bill Schilit, Norman Adams et Roy Want. *Context-Aware Computing Applications.* In First Workshop on Mobile Computing Systems and Applications, pages 85–90. IEEE, 1994. (pages 14 and 15.)

[Schreiber 2012] F.A. Schreiber, R. Camplani, M. Fortunato, M. Marelli et G. Rota. *PerLa: A Language and Middleware Architecture for Data Management and Integration in Pervasive Information Systems.* Software Engineering, IEEE Transactions on, vol. 38, no. 2, pages 478–496, 2012. (page 35.)

[Shelby 2014] Z. Shelby, K. Hartke et C. Bormann. *The Constrained Application Protocol (CoAP).* RFC 7252 (Proposed Standard), June 2014. (page 21.)

[Spinellis 2012] D Spinellis. *Don't Install Software by Hand.* Software, IEEE, vol. 29, no. 4, pages 86–87, 2012. (pages 91, 105, 116 and 162.)

[Stevenson 2010] A. Stevenson et C.A. Lindberg. New oxford american dictionary, third edition. Oxford University Press USA, 2010. evolution. (page 47.)

[Sudevalayam 2011] Sujesha Sudevalayam et Purushottam Kulkarni. *Energy harvesting sensor nodes: Survey and implications.* Communications Surveys & Tutorials, IEEE, vol. 13, no. 3, pages 443–461, 2011. (page 21.)

[Sun Microsystems 2006a] Sun Microsystems. *Java Development Kit 6 Documentation, Java JDK 6.* Documentation, 2006. (page 71.)

[Sun Microsystems 2006b] Sun Microsystems. *Java Management Extensions (JMX) Specification, version 1.4.* Specification, 2006. (pages 95 and 181.)

[Sun Microsystems 2013a] Sun Microsystems. *Enterprise JavaBeans, version 3.2.* Specification, 2013. (page 73.)

[Sun Microsystems 2013b] Sun Microsystems. *Java Servlet Specification, version 3.1.* Specification, 2013. (page 30.)

[Sun 2006] Yizhan Sun. *Complexity of system configuration management.* PhD thesis, Tufts University, jun 2006. (page 169.)

[Szyperski 2003] C Szyperski. *Component technology - what, where, and how?* In Software Engineering, 2003. Proceedings. 25th International Conference on, pages 684–693, 2003. (pages 48, 50 and 54.)

[Tajalli 2010] Hossein Tajalli, Joshua Garcia, George Edwards et Nenad Medvidovic. *PLASMA: A Plan-based Layered Architecture for Software Model-driven Adaptation.* In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, pages 467–476, New York, NY, USA, 2010. ACM. (pages 108 and 109.)

[Talwar 2005] V Talwar, D Milojicic, Qinyi Wu, C Pu, W Yan et G Jung. *Approaches for service deployment.* Internet Computing, IEEE, vol. 9, no. 2, pages 70–80, 2005. (pages 76 and 237.)

[Tigli 2009] JY Tigli, S Lavirotte, Gaetan Rey, V Hourdin, D Cheung-Foo-Wo, E Callegari et M Riveill. *WComp middleware for ubiquitous computing: Aspects and composite event-based Web services.* Annals of Telecommunications, vol. 64, no. 3, pages 197–214, 2009. (page 35.)

[Turnbull 2011] James Turnbull et Jeffrey McCune. Pro puppet. Apress, 2011. (page 106.)

[van der Burg 2011] Sander van der Burg et Eelco Dolstra. *A self-adaptive deployment framework for service-oriented systems.* In Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, pages 208–217, New York, NY, USA, 2011. ACM. (pages 108 and 109.)

[Waldner 2007] Jean-Baptiste Waldner et Jean Baptiste Waldner. Nano-informatique et intelligence ambiante: inventer l'ordinateur du xxie siècle. Hermès science publications, 2007. (pages 11 and 237.)

[Wang 2003] Nanbor Wang, Douglas C Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P Loyall, Richard E Schantz et Christopher D Gill. *QoS-enabled middleware.* Middleware for Communications, vol. 20, pages 131–162, 2003. (page 78.)

[Wang 2006] Xiaoning Wang, Wei Li, Hong Liu et Zhiwei Xu. *A Language-based Approach to Service Deployment.* In Services Computing, 2006. SCC '06. IEEE International Conference on, pages 69–76, 2006. (page 76.)

[Weiser 1991] Mark Weiser. *The computer for the 21st century.* Scientific American, 1991. (pages 12 and 13.)

[Weiser 1996] Mark Weiser et John Seely Brown. *Designing Calm Technology.* POWERGRID JOURNAL, vol. 1, 1996. (pages 10 and 11.)

[Zave 1997] P Zave et M Jackson. *Four dark corners of requirements engineering*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 6, no. 1, pages 1–30, 1997. (page 4.)