Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Autonomic Goal-Driven Deployment in Heterogeneous Computing Environments

Gabriel Siqueira Rodrigues

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientadora
Prof.ª Dr.ª Genaina Nunes Rodrigues

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenadora: Prof.ª Dr.ª Célia Ghedini Ralha

Banca examinadora composta por:

Prof.ª Dr.ª Genaina Nunes Rodrigues (Orientadora) — CIC/UnB
Prof.ª Dr.ª Célia Ghedini Ralha — CIC/UnB
Prof. Dr. Raian Ali — Bournemouth University

Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Autonomic Goal-Driven Deployment in Heterogeneous Computing Environments

Gabriel Siqueira Rodrigues

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof.ª Dr.ª Genaina Nunes Rodrigues (Orientadora)
CIC/UnB

Prof.ª Dr.ª Célia Ghedini Ralha   Prof. Dr. Raian Ali
CIC/UnB   Bournemouth University

Prof.ª Dr.ª Célia Ghedini Ralha
Coordenadora do Mestrado em Informática

Brasília, 16 de Dezembro de 2016

Abstract

We see a growing interest in computing applications that should rely on heterogeneous computing environments. In order to handle some kind of variability, such as two possible types of graphical processors in a desktop computer, we can use simple approaches as a script at deployment time that chooses the right software library to be copied to a folder. These simple ~~approaches~~ are centralized and created at design-time. They require one specialist or team to control the entire space of variability. Such approaches are not scalable to highly heterogeneous environments like Internet of Things (IoT), where each end user can have a different computing environment with broad range of different resources available. In such environments is impossible to predict the computing environment at design-time, implying ~~that deciding~~ on the correct configuration for each environment at design time ~~is impossible~~. In our work, we propose Goalp: a method that allows a system to autonomously deploy itself by reflecting about its goals and its computing environment. ~~We evaluate our approach based on an example of a case study. The algorithm for deployment planning returned the expected response for the tested case.~~

Keywords: dependability

Contents

List of Figures

Chapter 1

Introduction

Nowadays, people are surrounded by different devices with computing capability. Phones, watches, TVs and cars are example of daily devices for which there are smart versions with computing capability and where is possible to install software applications. Typically, these devices has connectivity capability and can form networks. These networks can be rich computing environment as each device brings different resources and capabilities. This present a great potential, but developing software that harvest the capability of such environment is very challenging. In this work, we call such environment a highly heterogeneous computing environment, a computing environment formed by different sets of devices, with different resources, and which are unknown at design-time. Ubiquitous Computing [8], Internet of Things (IoT)[5], Assisted Living[29] and Opportunistic Computing[45] are examples of domains that typically rely on highly heterogeneous computing environments for achieving user goals.

Software deployment is the process of getting a software ready to be used in a given computing environment[15]. It evolves planning which artifacts should be deployed, copying compatible artifacts to the target environment, configuring the environment and start execution. The *deployment planning* is a specially challenging activity, it requires analyzing the environment and the software architecture to solve variabilities, and come up with which software artifacts should be present in the deployment.

1.1   Problem Definition

Current software deployment approaches do not fit highly heterogeneous computing environment. The simplest approach to deployment as a whole is manual configuration, in which all steps in the deployment planning and execution are conducted by a human. It is normally applied when developing customized software that will be executed in devices managed by the development team. This approach do not scale for applications that target mass use, because it requires the deployment to be executed by a person with knowledge about the application internals. Another approach, common in cloud environments, is the use of scripts to automate software deployment execution. This approach is normally used in virtualized environments that simulate a very homogeneous environment. The scripts are tailored at design-time a specific target environment. When some variability can be solved at deployment-time with conditionals in the script, this is do not scalable as the script rely on a centralized model created at design-time. *Software store* is

1

another alternative approach. Typically, the developer uploads to the store the software configuration for each kind of target device, solving any variability at this point. In such cases, the deployment execution can relies actions to the ende-user such as accessing the store interface, searching for the application, and initiate the installation of the application. Neither scripts nor software stores are suitable for heterogeneous environments because they rely on a centralized method for deployment that requires knowledge about the target environment at design-time. In summary, current approaches for deployment do not suit deployment in highly heterogeneous computing environments as they require human interaction or knowledge about the runtime environment at design-time.

The challenges related to deployment in emerging highly heterogeneous computing environment can be summarized as follows:

- **Challenge 1: heterogeneity.:** the system is mean to run in a broad range of configurations of the computing environment.

- **Challenge 2: uncertainty at design time.** The system architech/developer do not know the configuration of the end user computing environment.

- **Challenge 3: deployment should be autonomous.** A deployment specialist probably will not be available in the target environment, so the deployment should be planned and executed autonomously

Many works have investigated the relation of goals and architecture of a system [50][39][30][38][40]. Some works in the literature has investigated variability at goal-models with adaptation purpose [53][4]. These works shows that goal-models are a promising approach to manage variability at the design of the software. But, to the best of our knowledge, none has investigated goal models at deployment level. Accordingly, our first research question emerges:

Research Question 1 (RQ1): Would a goal-driven approach be a viable one to manage variability at deployment?

With RQ1 we are interested in extend goal-oriented variability models to deployment level. With this we expect to allow the deployment of the system be adaptable to the characteristics of the target environment. However, in order to allow the adaptation, we also need to solve the variability, that is, we need to evaluate the points of variability of the system and the characteristics of the environment, and come up with a valid configuration that adapts the system deployment for the environment. From this, our second research question arises:

Research Question 2 (RQ2): Is it feasible and scalable to solve deployment variability autonomously, at deployment time?

With RQ2, we will investigate how to autonomously solve the variability, then finding a deployment plan that allows the achievement on goals in the target computing environment.

1.2 Proposed Solution

This work proposes ~~an approach – Goalp – that determines the computing environment and the available resources at runtime and to determine a suitable configuration from a general set of configurations for deployment in highly heterogeneous computing environments.~~ focus on autonomous deployment planning as the major part of the deployment in heterogeneous environments. In our approach, deployment planning occurs late in the software lifecycle, when the target computing environment is known. The planning is executed autonomously, do not requiring humans to interact with the system at deployment time.

An abstract model is used that consider the following information: (i) what the system needs to achieve (i.e., the goals), (ii) how it can achieve the goals (i.e., ~~its~~ its alternative strategies), and (iii) the resource restrictions. ~~The what~~ part (i) is ~~a~~ requirements model, ~~the how~~ part (ii) is facts containing software components and metadata, and ~~the restrictions~~ part (iii) ~~is~~ conditions that can be evaluated against the environment in order to find if a given artifact can be deployed.

Goal Oriented Requirements Engineering is a suitable modeling approach to model what the user want to achieve, where system requirements are modelled as intentions of actors in strategic goals[52][14][18]. Context goal models (CGMs) extend goal models[1], inserting the context as another dimension. We propose to use CGMs to model resource as context information that restricts how goals can be achieved, or more specifically which artifacts can be deployed.

Goalp consists of: (i) rules to refine context-goal model into software components, (ii) a description on how to create artifacts ~~that package components together~~ of relevant metadata; (iii) a metamodel that describes the deployment; (iv) an algorithm to analyse the metamodel and, for a given computing environment ~~and a~~ set of goals, select an appropriate set of artifacts that allows the achievement of the goals in the computing environment. The evaluation results show that the approach can be used to guide the development and the autonomous planning is able to plan the deployment of a system with thousands of artifacts in seconds.

1.3 Contributions Summary

This section summarizes the contributions of this proposal.

1. An approach to develop systems with variability at deployment

   - patterns to map components from a contextual goal model (CGM)
   - guide on how to package the components into artifacts keeping variability

2. An approach to autonomously plan the deployment at the target environment

   - A metamodel that describe the deployment
   - An algorithm to autonomously planning the deployment
   - A Java implementation of the algorithm

3

1.4 Structure

This dissertation is organized as follow. Chapter 2 introduces the ~~theory in which our work is based~~. Chapter 3 presents ~~related work~~. Chapter 4 Presents the case study Filling Station Advisor. Chapter 6 presents patterns and guidelines to develop software to heterogeneous computing environments. Chapter 6 presents the support for autonomous deployment. Chapter 7 depicts the evaluation of our ~~concepts~~ and Chapter 8 concludes the paper and outlines future works.

Chapter 2

Background

This chapter briefly reviews the concepts used throughout this work.

2.1 Context-aware Systems

Context-aware systems are ~~ones that are~~ able to adapt their behavior according to changing circumstances without user intervention. Finkelstein [21] describe a framework for context-aware services.
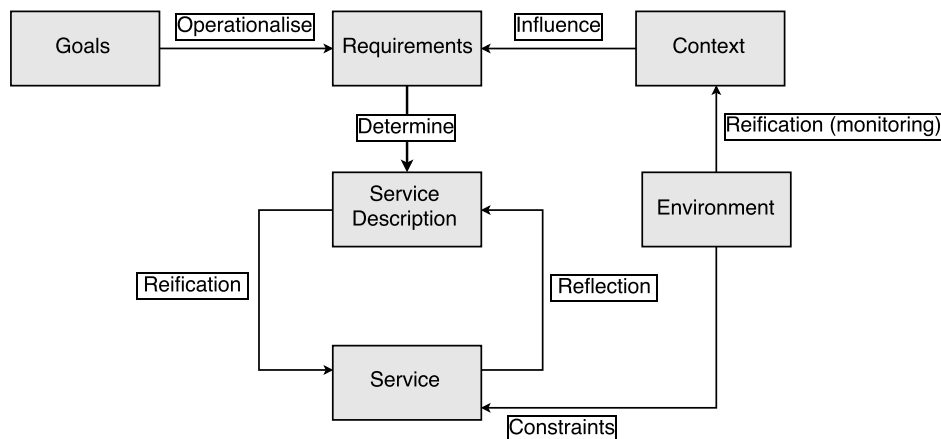
Figure 2.1: Context-aware services framework by [21]

Environment is whatever in the world provides a surrounding in which the agent is supposed to operate. The environment comprise such things as characteristics of the device that the agent is supposed to operate in. *Context* is the reification of the environment. The context provides a manageable, easily computer manipulable description of the environment. A context-aware system should match relevant environment properties and keep a runtime model that represents that information. By reasoning about that model the system can change its behavior. context can be either a *activator* of goals or a *precondition* on the applicability of certain strategy to reach a goal.

Goal is an objective the system should achieve. It is an abstract and long term objective of the system. A requirement operationalises a goal. It represents a more

5

concrete, short-term objective that is directly achievable through actions performed by one or more agents. Service description is the meta-level representation of the actual, real-world service. It should be a suitable formalism that allows services to be compared to requirements in order to identify runtime violations. Service provides the actual behavior as perceived by the user.

A *reflective system* is a system which incorporates structures representing (aspects of) itself. A *causal connection* between a model and a element modeled exists if one of them changes, this leads to a corresponding effect upon the other [32]. Following this approach, the system should keep a causal connection between the service and the description. The system adapts by manipulating the service description. Following a requirements reflection vision [11], a system should keep software requirements model at runtime, and use this model to drive the system adaptation.

2.2 Self-Adaptive Systems

Self-adaptivesses is an approach in which the system *"evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible."*[31]. Self-adaptive systems (SAS) aims to adjust various artifacts or attributes in response to changes in the self and in the context of a software system[43].

A key concept in self-adaptive systems is the awareness of the system. It has two aspects[43]:

- *context-awareness* means that the system is aware of its context.

- *self-awareness* means a system is aware of its own states and behaviors.

Schilit et al.[28] define *context adaptation* as "a system's capability of gathering information about the domain it shares an interface with, evaluating this information and changing its observable behavior according to the current situation".
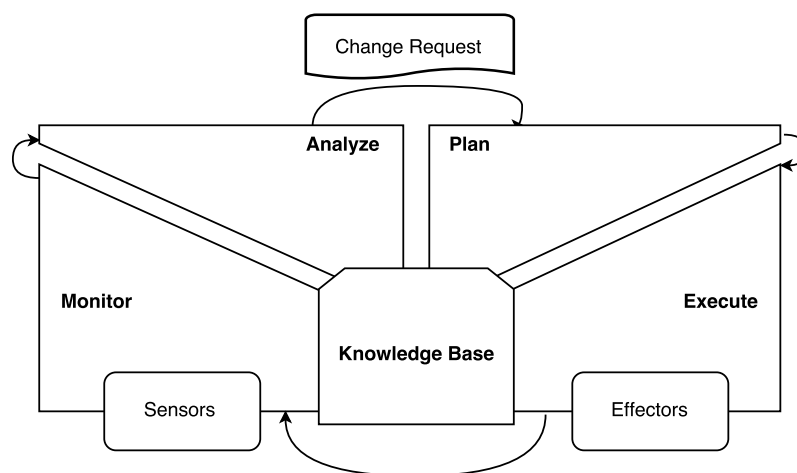

Figure 2.2: MAPE-K Reference Architecture

MAPE-K is a reference architecture originally proposed for autonomic computing [27] and that is often used as a model for architectures of SAS. It has a control loop, realized

6

by a simple sequence of four activities: *monitor*, *analyze, plan,* and *execute*. Where monitoring collects data from *sensors*, while executing is done through *actuators*. In addition, all activities can share information throught a *knowledge* system.

2.2.1 Development of Self-Adaptive Systems

For SAS, some activities that traditionally occur at development-time are moved to runtime. ~~In [3] was proposed~~ a process for development of adaptive systems. Activities performed externally are referred as *off-line activities*, and activities performed internally as *on-line activities*.
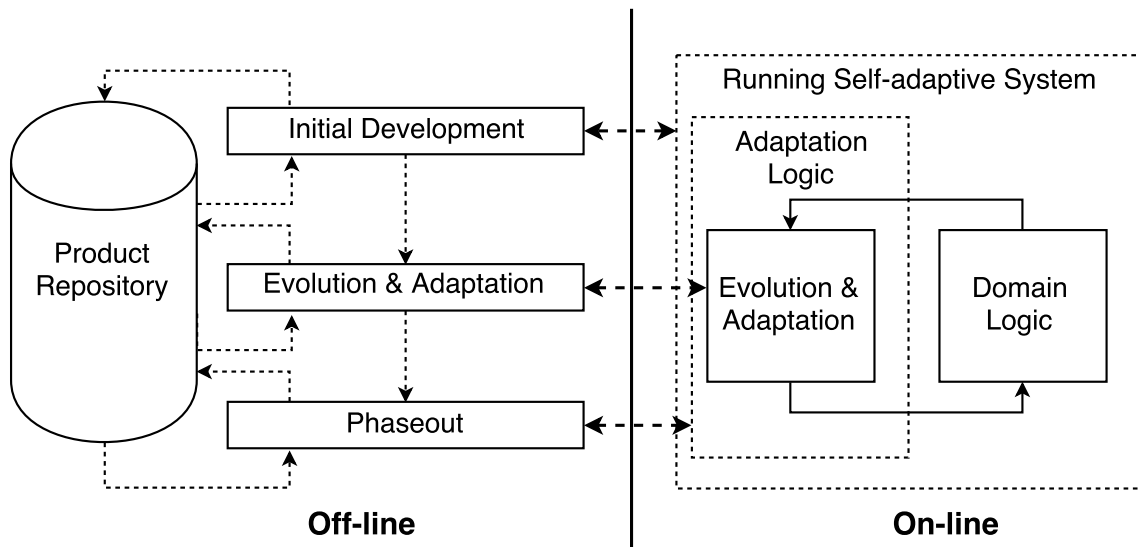
Figure 2.3: A Life-cycle model for Self-Adaptation Software System[3]

The right-hand side of Figure 2.3 depicts a running SAS. At this system we have *Domain Logic* that ~~solves that~~ is responsible for final user goals achievements. ~~And also,~~ *Adaptation logic*~~, that~~ is responsible ~~to adapt~~ the system in response to changes in the environment. Adaptation logic implements a control loop in line with the monitor-analyze-plan-execute (MAPE) loop [27].

The left-hand side of Figure 2.3 represents a staged life-cycle model. Off-line activities work on artifacts such as design model and source code in a product repository and not directly on the running system.

2.3 Dynamic Software Product Lines

Researcher have investigated dynamic software product line a way of adapt to a growing need for variations in users requirements and system environments.

DSPLs extend the concept of conventional SPLs by enabling software-variant generation at runtime. In classic SPL products can be derived from a SPL infrastructure for

a specific customer individual or customer segment, in the assumption that the requirements for that customer and the execution environment will not change. In DSPLs a product can change to another configuration, in runtime, in response to a context change. To make it possible the feature model should be available at runtime. [9]

DSPLs use the features models and orthogonal variability models (OVMs) as techniques for vability management, to model what are valid variabilities.

2.4 Goal Modeling

Goal-Oriented analysis is a requirements engineering approach that captures and documents the intentionality behind requirements. Goal Oriented Requirements Engineering (GORE) approaches have gained special attention as a technique to specify adaptable systems [37]. Goals capture the various objectives the system under consideration should achieve. In particular, Tropos[14] is a methodology for developing multi-agent systems that uses goal models for requirement analysis. Wooldridge [51] defines Multiagent Systems (MAS) as systems composed of multiple interacting computing elements known as *agents*. Agents are computer systems that are capable of autonomous action and interacting with other agents. The Jadex is a platform that facilitate development of mult-agent systems [13].

The Tropos key concepts

Tropos use a modeling framework based on i* [52] which proposes the concepts of actor, goal, plan, resource and social dependency to model both the system-to-be and its organizational operating environment [14] [36].

In Tropos, requirements are represented as actors goals that are successively refined by AND/OR refinements. There are usually different ways to achieve a goal, and this is captured in goal models through multiple OR refinements.

Key concepts in the Tropos metamodel are:

Actor an entity that has strategic goals and intentionality

Agent physical manifestation of an actor.

Goals it represents actors' strategic interests. *Hard goals* are goals that have clear-cut criteria for deciding whether they are satisfied or not. *Softgoals* have no clear-cut criteria and are normally used to describe preferences and quality-of-service demands.

Plan it represents, at an abstract level, a way of doing something. The execution of a plan can be a means for satisfying a goal or for *satisficing* (i.e. sufficiently satisfying) a softgoal.

Resource it represents a physical or an informational entity.

Dependency it is a relationship between two actors that specify that one actor (the *depended*) have a dependency to another actor (the *dependee*) to attain some goal, execute some plan or deliver a resource. The object of the dependence is the *dependum*.

Capability it represents both the *ability* of an actor to perform some action and the *opportunity* of doing this.

In Tropos requirements are represented as actors goals that are successively refined by AND/OR refinements. There are usually different ways to achieve a goal, and this is captured in goal models through multiple OR refinements.

Goal models are a traditionally a requirements tool, as such it must capture the space of the solution and are not sufficiently detailed to reason about system execution and do not capture information on the status of requirements as the system is executing, nor on the history of an execution [12]. These traditional goal model can be named design-time goal model (DGM). Dalpiaz et al.[17] describe a method for extend Design-time Goal Models (DGMs) to create Runtime Goal Models (RGM). RGMs can be used to analyze the system's runtime behavior. Other works relates goal models with another dynamic aspects of systems, such as configuration [53], behavior [17], probability of achieving success [34] and achievability of goals [41].

Salehie et al. [44] propose a run-time goal model and its related action selection. It models adaptable software as a system that exposes sensors and effectors and proposes a model consisting in Goals, Attributes and Action for selecting actions that will effect the adaptable software at runtime, giving sensed attributes. So the adaptation mechanism is to choose the best action given the actual attributes. It uses explicit runtime goals and make them visible and traceable.

Mendonça et al.[34] propose a methodology for fault forecasting by which developer, at design time, annotate the goal decomposition in goal model and specify context variables.A special tool generate a formula for, given a context, evaluate the probability of achieve a goal at runtime.

Contextual Goal Model

Contextual Goal Model, proposed in [1], captures the relation between system goals and the changes into the environment that surround it. Context goal models extends goal models with context information. Goals and context is related by inserting context conditions on variation points of the goal model. Context Analysis is a technique that allows to derive a formula in verifiable peaces of information (facts). Facts are directed verified by the system, while a formula represents whether a context holds.

2.4.1 Software Deployment

Software deployment refers all the activities that make a software system available for use[15]. These activities results in the creation and distribution of artifacts, from the development environment to the target runtime environment. Artifacts are files that package software components and assets. The deployment process can vary depending on the application domain and execution platform. In embedded platforms the deployment can consist in burn software into a chip. In consumer personal or business domain, for a desktop platform, the deployment can consist in a install process with collaboration between a person and a script that automate some steps. In a enterprise domain, for a web platform in can consist in coping and editing some files in a couple of machines. In many of this scenarios software will be periodically updated, frequently being unavailable

in the process. The complexity of the software deployment can also vary in function of how much the platform is distributed (i.e. the number of nodes), how much heterogeneous it is, and how much is known about the deployment computing environment at design-time. In a dynamic and heterogeneous environment deployment can be specially complex.

Artifacts

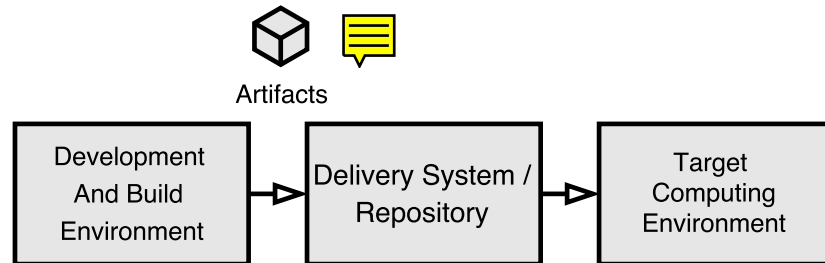| Development And Build Environment	→	Delivery System / Repository	→	Target Computing Environment

Figure 2.4: Artifacts Deployment

Deployment artifacts are the artifacts needed at the deployment environment. Artifacts are build at development and build environment. Built artifacts are move for a delivery system where they can be accesses from the target environment. At deployment the artifacts are moved from the delivery system to the target computing environment. Also configuration activities can be realized. In the software industry, a *continuous integration* environment applies automation in building and getting components ready to delivery. In such a environment if a developer push changes to a code repository components are automatically built and published to delivery system. The build process commonly evolves fetching build dependencies, compiling source code, running automated quality control (tests and static analysis) and packaging. Artifacts are published if target quality policies are met. Fundamental to continuous integration environments are *Package Management Systems* tools. Theses tools simplify the process of manage software dependencies [46]. That tools ensure that development team members are working with same dependencies that are used in the build environment.

Research in software configuration and deployment, has focused on respond to dynamisms in a known environment. This could be costs and failures in a cloud environmet [20], changes in managed resources [24], and changes in the context of operation [10].

Continuous delivery extends the continuous integration environment, moving components from the delivery system to a target computing environment with none or minimum human intervention. In modern enterprise environments the deployment activity has seen significant changes with popularization of use of virtual machines and cloud computing.

In the industry, package managers such as aptitute/apt-get(Debian based Linux distributions), yum (Red Hat based Linux distributions), Homebrew (MacOS) and Chocolatey (Windows) are capable of solve dependencies and deploy software. They require that a managed application declare their dependencies by name and version. Devops[6] is a movement in software industry that advocates that all configuration steps needed to configure the computing environment should be written as code (*infrastructure as code*), and follow best practices of software development. That movement favor the documentation, reproducibility, automation and scalability. Devops allow for management of scalable computing environments. It can offer significant advantage for enterprise environment in relation to manual approaches in which system administrators configure the system by

manually following configuration steps. Current continuous integration/delivery and devops practices are not sufficient for highly dynamic and heterogeneous target computing environments; they requires that highly specialized system administrators to analyze the environment and create environment configuration descriptors.

2.5 Software Components

Heineman define *software component* as a "software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard"[25].

Software components is a unit of composition. Software systems are build by composing different components. Software components must conform to a component model by having contractually specified interfaces and explicit context dependencies only.[48].

Component based software engineering (CBSE) approach consists in building systems from components as reusable units and keeping component development separate from system development[16].

A *component interface* "defines a set of component functional properties, that is, a set of actions that's understood by both the interface provider (the component) and user (other components, or other software that interacts with the provider)"[16]. A component interface has a role as a component specification and also a means for interaction between the component and its environment. A *component model* is a set of standards for a component implementation. These standards can standardize naming, interoperability, customization, composition, evolution and deployment.[25] The *component deployment* is the process that enables component integration into the system. A deployed component is registered in the system and ready to provide services[16]. *Component binding* is the process that connects different components through their interfaces and interaction channels.

Software architecture deals with the definition of components, their external behavior, and how they interact[26]. The architectural view of a software can be formalized via an architecture description language (ADL)[33].

Component based software engineering (CBSE) approach consists in building systems from components as reusable units and keeping component development separate from system development[16].

CBSE is built on the following four principles[16]:

- *Reusability.* Components, developed once, have the potential for reuse many times in different applications.

- *Substitutability.* Systems maintain correctness even when one component replaces another.

- *Extensibility.* Extensibility aims to support evolution by adding new components or evolving existing ones to ext the system's functionality.

- *Composability.* System should supports the composition of functional properties (component binding). Composition of extra functional properties, for example composition of components' reliability, is another possible form of composition.

2.5.1 Component-Based Adaptation

In the literature was proposed frameworks for architecture and components based adaptation.

Rainbow[23] is a framework for self-adaptation architecture based. It keeps an model of the architecture of the system and can be extended with rules to analysis the system behavior at runtime, find adaptation strategies and perform this changes. It separate the functional code (internal mechanisms) from adaptation code (external mechanism) in a schema called external control, influenced by control theory.

MUSIC[42] project provides a component-based middleware for adaptation that propose to separate the self-adaptation from business logic and delegate adaptation logic to generic middleware. As in our propose it adapts by evaluating in runtime the utility of alternatives, to chose a feasible one (e.g., the one evaluated as with highest utility).

Flashmob [47] is an approach for distributed self-assembly. Different from MUSIC and Rainbow, it handles component-based adaptation in a distributed environment. The self-assembly can be described as: given a set of available components (with various functional and non-functional properties), and a configuration of components which are already running, find a new configuration which works (better) in the changed execution environment (including hardware), meets new user requirements or takes account of new component implementations [47]. Flashmod uses a three-layer model: goals, management and components proposed by Kramer and Magee [30], extending it to allow distributed agreement in a given configuration.

OSGi[49] is a Java centric platform that allows dynamic bind and unbind of components, usually named bundles. Ferreira et al.[19] proposed a framework for adaptation based on OSGi.

2.5.2 From Goals to Components

Lamsweerde [50] present a method for derive architecture from KAOS goal model. First an abstract draft is generated from functional goals. Secondly, the architecture is refined to meet non-functional requirements such as cohesion.

Pimentel et al. [40] present a method using i* models to produce architectural models in Acme. If focus in he development of adaptive systems. First, it transforms a i* model into a modular i* model by means of horizontal transformation. Secondly, it creates an architecture model from the i* modularized model by means of vertical transformation. Architectural design models is made easier by the presence of actor and dependency concepts.

Yu et al. [53] proposed an approach for keep the variability that exists in the goal model into the architecture. It present a method for creating a component-connector view from a goal model. A preliminary component-connector view is generated from a goal model by creating an interface type for each goal. The interface name is directed derived from the goal name. Goals refinements result in implementation of components. If a goal is And-decomposed, the component has as many *requires* interfaces as subgoals.

```
Component G {
  provides IG;
  requires IG1, IG2;
```

```
}
```

If the goal is OR-decomposed, the interface type of subgoals are the interface type of the parent goal.

```
Component G1 {
    provides IG;
}

Component G2 {
    provides IG;
}
```

A component equivalent to the parent goal is generated with a switch.

Dependency Injection

Dependency Injection is a pattern that allow for wiring together software components that was developed without the knowledge about it others[22]

In OO languages normally you instantiate an ~~Object~~ from a class using an operator (*new* for Java) and a reference to ~~this~~ class. ~~By this the object that is instantiating~~ (the object client) is dependent of the referenced class (the service implementation class). In case of strong typed languages, normally one will get an exception if the referenced class is not present.

So the use of the *new* operator lead to the following disadvantages:

- impose compile time dependency between two classes

- impose runtime dependency between two classes

The basic idea of the Dependency Injection, is to have a separate object, an assembler, that wire together the components at runtime[22]. The client class refer to service using its Interface (the service interface). The assembler can use alternative ways of the *new* to instantiat~~ing~~ an object, that the wiring between client objects and implementation service classes could be postpone to runtime.

By this is we can at runtime:

- discover available implementation of a service interface

- decide this implementation to instantiate from available implementations

Not always this pattern is needed as not always is useful to avoid the reference to a class. But in the context of component-based adaptation it would be specially useful to the decouple client components from service components, allowing runtime reasoning about what implementation to choose.
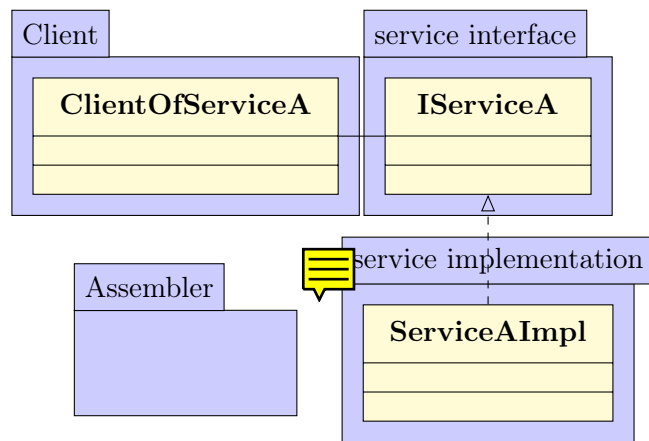
13

Figure 2.5: Two components

Chapter 3

Related Work

In this chapter, we ~~will~~ highlight ~~the~~ our most closely related work. Table 3.1 ~~summarize research related to Goalp.~~

Bencomo et al. [10] uses a SPL approache to adaptation. It associates an architecture variability model with an environment variability model. The environment variability is modeled as a transition system. The structural variability is responsible for the system adaptation. A configuration or a product is a set of component selected. A configuration is associated with states in the environment variability model. Unlike our approach, their focus is on the adaptation in the configuration at runtime but not in the deployment. [35] ~~uses~~ a feature model associated with context requirements.

Leite et al. [20] proposes an approach for automatic deployment on inter-cloud environments. It relies on abstract and concrete features models and constraint satisfaction problem solver to create a computing environment using resources distributed across various clouds. It integrates a self-healing schema for cloud deployment based on which virtual machines in the cloud are monitored and in case of failure the machine can be restarted or terminated and them a new one created. The approach is specific to cloud environments and require instantiating at design-time a model ~~with knowledge~~ about the environment. It also strongly depends on design-time created scripts to realize the deployment of an application, which limits the autonomy of the approach, specially in ~~a~~ unknown environment.

Gunalp et al. [24] presents an approach for automatic deployment, in which the deployment specialist specifies the system deployment in terms of resources and desired target states of ~~this~~ resources. The approach follow predefined strategies to keep the managed software resources in the specified states. They uses a low level model to driven the adaptation (target states and strategies to change states). Differently, our approach uses a goal-model with is a more abstract model.

Angelopoulos et al. [4] present an approach to handle variability at three different dimensions: goals, behavior and architecture. Variability can occur at goals dimension as an OR-refinement or context selection; at behavior dimension as different plans flows; and in architecture with variability of components and implementations. However, their approach does handle variability at deployme.

Ali et al.[2] explore the optimization of the deployment for a given context variability space in which the system will be deployed. CGM was used to represent aspects of the environment related to the ~~space of solution~~, that were to be analyses at design-time. This

analyses at design-time can be used to evaluate which alternative strategy to implement. It differ from our work in which we explore the context of the computing environment. Our approach ~~analysis~~ allow for choosing between components already implemented at deployment time. Both approaches could be used in tandem, as they ~~use the same model (CGM) and provides different analysis.~~

Package managers such as aptitute/apt-get are capable of solve dependencies and deploy software. They require that a managed application declare their dependencies by name and version. In our approach, ~~differently,~~ the dependencies are declared in terms of interfaces for which implementations are required, not specific implementations. ~~For example, in our case study, the advisor root strategy declares that it requires any artifact that provides *Get Position* goal achievement, but not a specific artifact is required. This requirement declaration, in terms of interface,~~ associated with context conditions allow for a more flexible dependency resolution at deployment-time.

Table 3.1: Comparing characteristic properties of selected approaches related to Goalp

Work by	Goal Oriented	Handle Heterogeneity	Autonomic Deployment Planning
Ali et al.[2]	Yes	No	No
Angelopoulos et al. [4]	Yes	No	No
Mizouni et al. [35]	No	Yes	No
Leite et al. [20]	No	Yes	No
Gunalp et al.[24]	No	Yes	No
Goalp	Yes	Yes	Yes

Chapter 4

Filling Station Advisor

4.1 Motivating Example: The Filling Station Advisor

In this paper, we use a case study of a filling station advisor application. Filling station here refer to a place where the car can be refueled or recharged (gas station/petro or charge station). The main goal of the filling station advisor is to give direction to a vehicle driver about nearby filling stations that can be reached conveniently. By convenient we mean that certain conditions for the chosen station have to be fulfilled as well as user preferences are considered. Examples of conditions are: fuel is compatible with the vehicle; station is located inside the vehicle distance-to-empty. Examples of users preferences are: low price, low number of stops, small deviation from an actual route, and station reputation.

In this work, we will focus on the challenge of handling the computing variability when developing such application. To maximize the utility, the filling station advisor should be able to run in a broad range of devices like smart-phones and car navigation systems. Each of such devices can have a different set of resources that can be used to find a convenient filling station according to the user preferences. For example, in a scenario where a human driver is using the application with a smart phone, we could use the GPS resource to track the position and the distance since the last refueling; the Internet connection to find nearby filling stations; the device text-to-speech engine to create a voice message to alert the driver when he is passing by a convenient filling station. In another scenario, in which the application is running in on-board computer of an Internet connected self-driving car, we could use a more precise distance-to-empty data from on-board computer, and replace the text-to-speech notification with a system call to the vehicle self-driving system advising the next filling station stop.

The filling station advisor's main goal can be refined into the following five goals, each one with its own computing resources requirements:

Identify Vehicle Position: the system should identify the vehicle position using an available positioning system. To fulfil such goal, a GPS or cell antenna triangulation could be used.

Assess Distance to Empty: the system should make use of the best available data about the vehicle distance to empty. It could be: access a standard or proprietary interface within the vehicle that provides the data directly as calculated by the on-board computer; use an interface to access data about fuel level and mileage

average and calculate the distance to empty; ~~use~~ user input ~~about~~ tank capacity, vehicle mileage, and keep track of ~~distance traveled~~ since the last ~~time the tank was felt completely.~~

Recover information about nearby filling stations: the system should recover information about nearby filling stations ~~by:~~ querying available services on the Internet, if connection and servers are available. Otherwise, the system should use previously cached results.

Decide on the most convenient filling station: Based on position, distance to empty and nearby gas stations, the application should try maximize some user preference, it being low cost, low number of stops, ~~prioritize an automotive~~ fuel brands or gas station reputation.

Notify Driver: the application should decide when and how to notify the driver with advices on when to stop in a filling station. The notification could be integrated with an active navigation system if such an interface exists; otherwise it should notify the driver using text-to-speech engine, a pre-recorded voice audio, or on-screen notification.
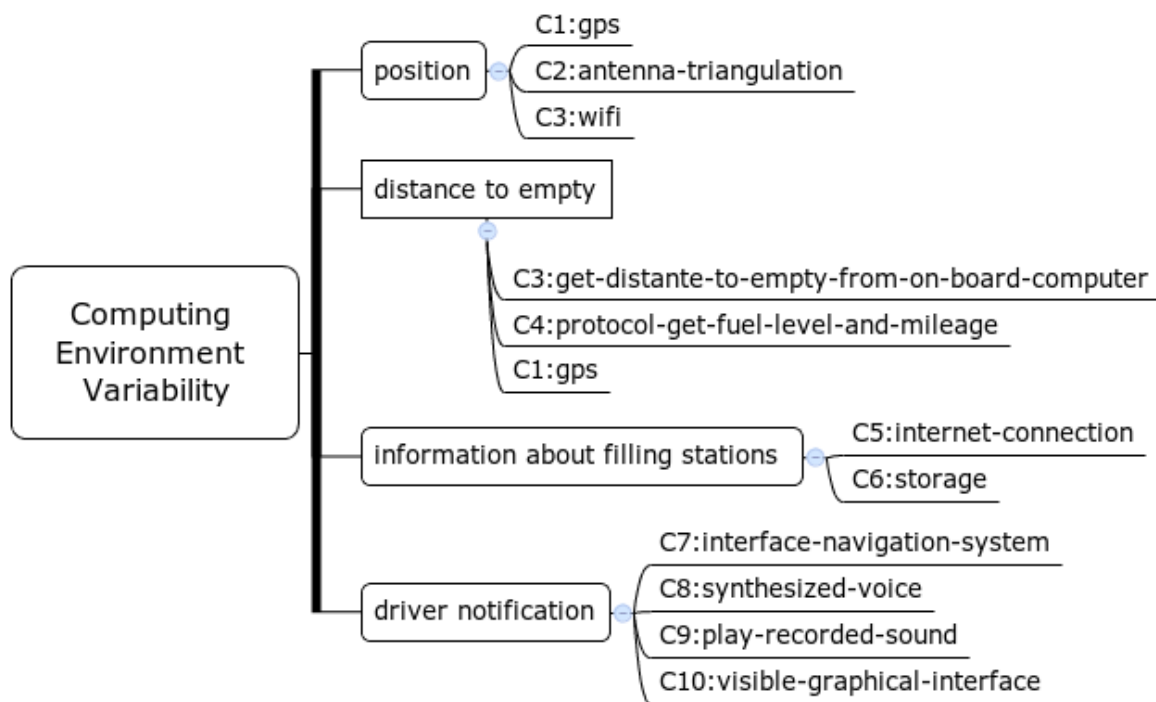

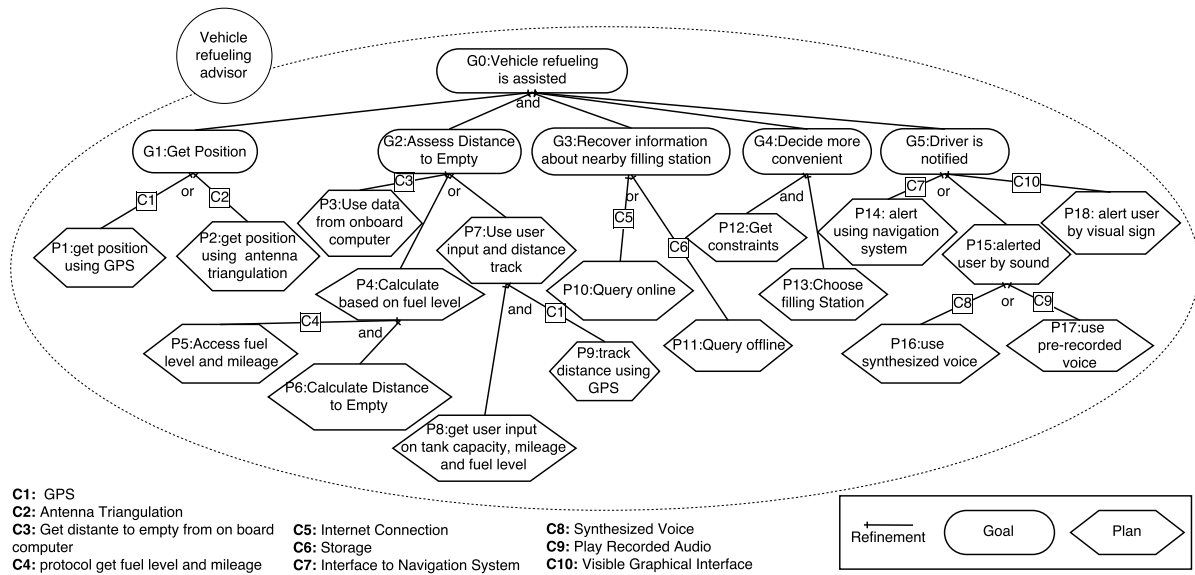Figure 4.1: Variability in the Computing Environment

Figure 4.2: CGM of the filling station advisor

Chapter 5

From Goals To Artifacts

5.1 Proposed Approach

Following the model proposet by Andersson et al.[3] for adaptive software development process, we divide our approach into *offline* and *online* activities. In this work, the *offline* activities are conducted by software engineers, and result in development and publishing of software components. The *online* activities are autonomously executed in target environment, and result in the deployment of the system. Figure 5.1 presents an overview of the process.
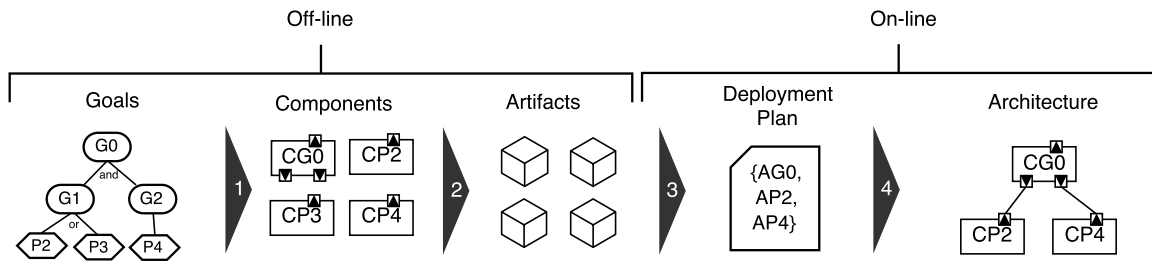

Figure 5.1: Activities: (1) component mapping; (2) packaging; (3) deployment planning; (4) component binding

In the artifact creation process, components are mapped from the system CGM using patterns. These components are packaged into artifacts together with their metadata, which describes what goals the artifact provides, its context conditions and dependencies. Next, the artifacts are put into a repository. As the result of the offline activities, artifacts are created that implement a system specified by a goal model.

In the online part of the approach, the Goalp deployment planning is executed in the target computing environment and autonomously finds a deployment plan. The deployment planning is responsible to look into the repository of artifacts and based on context information and goals, find a set of artifacts that should be deployed to the target computing environment in order to make the goals achievable.

Previously, goal-driven approaches was proposed for introducing variability at requirements, context modeling, software behavior, and software architecture[53][4]. In our

methodology, we propose a systematic approach to support deployment variability, from requirements to deployment. Deployment variability is important since not taking into account the heterogeneity of a computing environment may lead to unnecessary or even unsuited deployment of components. Such scenario would bring a negative impact to software performance, or in some cases represent inconsistent deployment of functionalities on the target device.

When developing a monolitic software, we implement in the same codebase all functionalities, then all code is build and deployed together. In the Filling Station Advisor example, if implementing it as a monolic software, the logic to get the vehicle position using GPS or antenna triangulation would stay in the same codebase and would be deployed altogether in the target environment, even when it does not have antenna triangulation capability.

In order to better cope with heterogeneity in the computing environment we should minimize the coupling between parts of the code that have dependencies of specific resources in the environment. By encapsulating, into components, dependencies on specific resources, it is possible to create variability at architecture level. By packaging the components into different artifacts, it is possible to maintain such variability at deployment level. This variability is useful as it allows the deployment of components only to environment that has the required resources.

Regarding the Filling Station Advisor example, components can be implemented providing the actual position of the device by means of GPS or antenna triangulation. These components can be packaged into different artifacts that will only be deployed when the target environment has the appropriate resources.

5.1.1   CGM for heterogeneous computing environments

A systematic way of analysing the capabilities of the computing environment is needed in order to support the resolution of variability at deployment-time. First, the available capabilities in the environment should be represented in the context.

Definition 1 (Resource) *A resource provides a specific computing capability, it could be available in the computing environment and used in plans. A resources receive a label.*

Definition 2 (Context) *A context Ctx := r1..r, { r ∈ Ctx | a resource labeled r is available in the computing environment}*
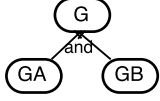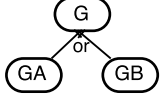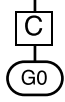
Context is a set of labels. In our example, the set [gps-capability, internet-connection, synthesized-voice] is a an example of context, representing that GPS, connection to the Internet and voice synthesizing are resources available in the computing environment.

Then, the applicability of a plan should be related with available resources. Conditions represent restrictions related with the environment, and can be evaluated against the context.

Definition 3 (Context condition) *A context condition cx:=TRUE iff cx:r, r ∈ Ctx*

A context condition is satisfied if the associated resource is present in the context. In a scenario with context Ctx=[gps, internet-connection], the context condition c1:gps holds.

Table 5.1: Contextual Goal Model to components; (1) And-refinement, (2)OR-refinement, (1) context condition,

(G → and → GA, GB)	Component CG {
	   provides IG;
	   requires
	      IGA, IGB;
	 }

(G → or → GA, GB)	Component CG {
	   provides IGA;
	 }
	 Component CG {
	   provides IGB;
	 }

(C → G0)	Component CG {
	   provides IG;
	   condition C;
	 }

As proposed by Rain at al [1], context conditions are used to solve variability. In Figure 4.2, the goal G1 has two alternatives to be achieved: by executing plan P1 or P2. The plan P1 is applicable if the context condition c1 holds, which is the case when GPS is available in the environment.

5.1.2 Preparing Components

Goals to Components

Components are architectural units. In our proposal, components definitions are mapped from the CGM and them developed by the architect/developer.

The patterns present in table 5.1 are used to map components based on the CGM of the system. By mapping components we mean identifying which component should be developed in order to reflect the CGM of the system. By using the proposed patterns, the variability present in the CGM is kept at architecture of the system. Theses patterns are an extension of Yu et al.[53] patterns for the Goals-Component view, with contextual conditions.

The presented patterns are described using goals, but they can be applied for goals and plans, without distinction.

And-refinement result in components that define a strategy to achieve a given goal by achieving two or more sub more concrete goals. Mapping components from a Goal And-refinemnt result in: (i) a root interface that describe what component provides. (ii) Interfaces for each sub goal. (iii) A component that provides the interface (i) and requires each interface generated for sub goals (ii). That component (iii), implements a strategy to achieve its provided goal. It coordinates the sub more specific goals by calling then, and passing one result as input on another, when applicable. As an example, applying And-refinement patterns for Root Goal G0 of the Filling Station Advisor application, will result in interface IG0 and a component G0 that provides IG0 and requires IG1, IG2, IG3, IG4, and IG5.

```
Component G0 {
    provides IG0;
    requires IG1, IG2, IG3, IG4, IG5;
}
```

Applying or-refinements pattern, results in a root interface definition and in multiple implementation. When Or-refinements is associated with context-conditions, it allows for alternative strategies using different resources in the computing environment. For example, in the Filling Station Advisor, applying the patterns for G1, P1 and P2 will result in the following components:

```
Component CP1 {
    provides IG1;
    condition C1;
}
Component CP2 {
    provides IG1;
    condition C2;
}
```

The two plans P1 and P2 associated with the Or-refinement has different context conditions (C1:gps and C2:antenna triangulation). That variability in the design allows for the adaptation to the heterogeneity in the target environment.

Artifacts

From the deployment point of view, the components and interfaces should be packaged in an file archive to distribution. We name such file an artifact. An artifact should follow a standard packaging schema, so it can be manipulated by a package manager. In our approach, we propose to include into the artifact metadata that describe the artifact provided goals, context conditions, and dependencies. That metadata reflect information about the packaged components. For our approach, the metadata of interest is the following:

Provided goals: goals that can be made achievable by successfully deploying the component.

Context conditions: conditions that can be evaluated against the context. If the conditions are not satisfied it means that the component can not be deployed at the

given context. This is the case when the artifact required resources are not available in the computing environment.

Dependencies: required goals that should be provided by other artifacts.

When creating the artifact we can calculate the metadata by looking at the packaged components. The artifact *provided goals* metadata are the union of all *provided goals* of the components packaged in this artifact. The same is valid for *context conditions* and *dependencies* metadata.

Both *context conditions* and *dependencies* impose restrictions on when an artifact can be deployed. Context conditions refer to the need on resources in the computing environment that are beyond the deployment agent capacity of management. Such dependency can be related to hardware implementation, e.g. GPS-module, our platform lower level software implementation and access authorization, e.g. access to vehicle on-board computer data. If a context condition do not hold there is nothing that can be done at deployment time to change that. Dependencies, differently, refer to the need on another artifacts. It is specified in the terms of Provided Goals. Like another dependency management approaches, an artifact can depends on another artifacts. Different from other approaches, we specify the dependency not to a specific version of an artifact identification but in terms of what an artifact provides. So, in the Filling Station Advisor, the artifact A0, that packs the component G0 depends on IG0-definition and IG0. To satisfy the dependency an artifacts that provides IG0-definition should be available, as well as at least one artifact that implements IG0.

Artifacts are registered in a repository which allows the distribution of artifacts to target environment. In the registration process, the artifact is upload to the repository, its metadata is read, and registered in the repository database.

Deployment Architectural Style

In order to maximize the flexibility of systems implemented using Goalp deployment approach, is proposed an architectural style for creating artifacts.

Artifacts can have be of 3 types:

Definition artifacts that specify the interface of goals. It contains interfaces declarations and data model. It specify the API or contracts for a given set of goals. The advantage of separating the goals declaration in a specific artifact is creating implementation independence. Goals declarations depends only on other goals declarations and have no context conditions. Goals declarations do not provide any goals.

Strategy artifacts that package components that result from AND-refinements. A Goal refinement provides a high level goal and depends on other more refined goals. It should have no context condition as it do not implement plans that use specific resources in the computing environment.

Plan implementation theses artifacts contain the domain logic implementation. The plan could have dependencies on specific resources on the computing environment.

In a dependency tree we have plan implementation artifacts as leafs. Plan implementation artifacts provides low level goals and can have dependencies and context conditions.

These types of components has not different treatment by the deployment planning but following it will increase the flexibility of deployment.

5.2 The Process

Roles

The proposed process considers three roles: users, requirements engineers and software architects. Figure 5.2 summarize the collaboration between the roles.
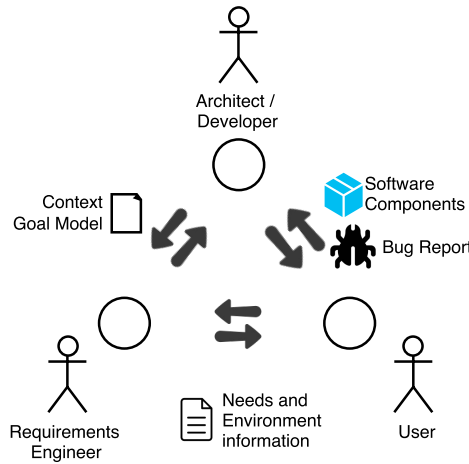

Figure 5.2: Roles collaboration

User This role has access to a particular computing environment and want to achieve some goals there.

Requirements Engineer Is responsible to translate users goals to a contextual goal model. Also is responsible to analyze the different contexts that the system is meant to operate and how it affects the goals.

Architect Architect project the software architecture such as to permit variability of deployment. From the point of view of dynamic heterogeneous computing environments, the focus is to create interfaces for components that can allow for goal achievements using different computing resources.

Activities

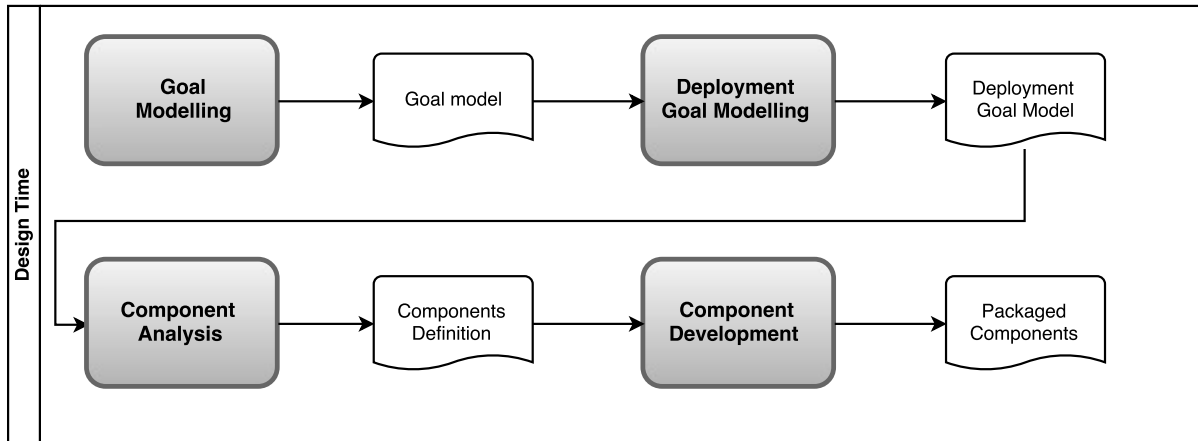Figure 5.3 describe the development process activities.

Figure 5.3: Deployment Process Activities

Goal Modeling

This phase is coordinated by a requirement engineer with participation of a domain specialist, possibly the user. A process such as TROPOS can be used. The output of this phase is a goal model. Here, the goal model assumes a central role in the software development process. The goal model besides define the space of the solution, also works as common language. The goal model formalize the domain knowledge. It defines a common language between user and developers that will allow software deployment driven by user goals.

Context Goal Modeling

In this phase, the Goal model should be annotated with *context conditions* related with the computing environment. That analysis is a context analysis and could benefit of the process described in [1]. The requirements engineer should use knowledge about the computing resources that may be available at the computing environment.

Component Analysis

Software engineer should identify variability points. Variability points in the contextual goal model are points where goals can be achieve with different strategies, each one having different context conditions. Component interfaces are created following the guidelines described in Section 5.1.2. The input and output of components are defined.

Also at this phase, it should be defined the sensors needed to evaluate facts about the computing environment.

Component Development and Packaging

Component development includes the cycle coding, build and test of software components. The component package in the standard packaging schema is an artifact and should be put in a delivery system.

Chapter 6

Autonomic Deployment Planning

The deployment planning is a online part of the presented approach, executed in the target environment. It is conducted autonomously, using a metamodel and algorithm to come up with a deployment plan.
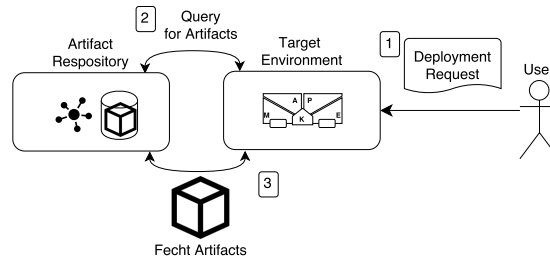

Figure 6.1: Goald Deployment Actors

Figure 6.1 depicts the deployment execution. A user interested in using a computing environment to achieve a set of goals submits to this environment which goals it want to achieve in the form of a deployment request.

Then, the system introspect about available computing resources and artifacts present in repository and plan the deployment, generating a deployment plan that is a selection of artifacts that can allow for the goals achievement in the available computing environment. The deployment is them executed by fetching the appropriate artifacts from the repositories.

6.1   Deployment Planning

Figure 6.2 presents the metamodel used. Artifact is the central entity at deployment level. As described in the section 5.1.2, artifacts has *provided goals*, *context conditions*, and *dependencies*. Artifacts *provided goals* and *dependencies* create relations of dependency between artifacts, so that an artifact that has a goal dependency is dependent on an artifact that provides that goal.

An *agent* can accept deployment requests, action that should trigger the deployment planning. An agent knows a *respository* where it looks for artifacts. A *repository* has a set of artifacts that it can be queried about by the *queryForArtifacts* method. The method
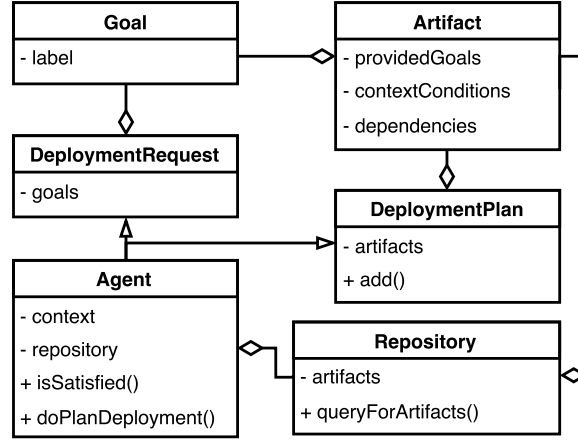
Figure 6.2: The Goalp Deployment metamodel

queryForArtifacts receives a Goal as argument and return all artifacts in the repository that provide that Goal. An *agent* can verify artifacts *context conditions* satisfaction against its own context by *isSatisfied* method.

The *Deployment Request* is a set of goals that an external entity sent to an agent, requesting it to plan a deployment. The *Deployment Plan* is generated as a result of agent *doPlanDeployment*. The *Deployment Plan* is a set of artifacts that makes the goals specified in the *Deployment Request* achievable.

Note that components do not appear here in this model. Components are architectural units that are packaged into artifacts. The components definitions are mapped and developed by the architect/developer, offline. And is instantiated and bind by the platform, online. But, it do not appear directly at deployment reasoning, as the abstraction concept at deployment is the artifact.

Artifacts are *deployable* for an agent if all its context conditions and dependencies are satisfiable. Goals are *achievable* if artifacts that provide that goal are deployable, so there is a *deployment plan* that is able to satisfy this goal.
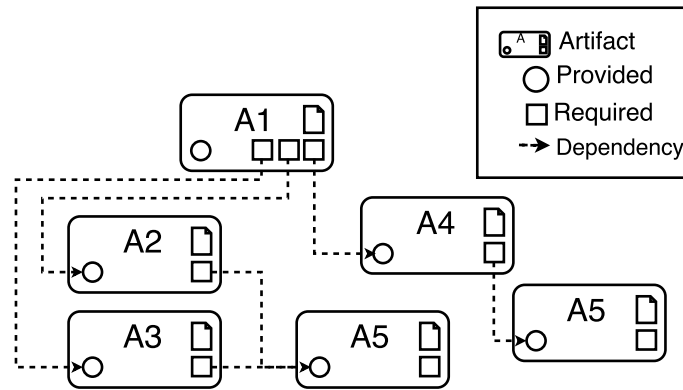

Figure 6.3: Dependency Graph

6.1.1 Planning Method

To come up with a deployment plan for a given a given deployment request and context we present the Algorithm 1. It implements the *Agent*'s *doPlanDeployment* method (Figure 6.2).

Input: DeploymentRequest request
Result: DeploymentPlan plan

```
 1  var resultingPlan ← new DeploymentPlan()
 2  foreach Goal selectedGoal in goals do
 3  │   var subPlan ← new DeploymentPlan()
 4  │   var artifacts ← repository.
 │   queryForArtifacts(selectedGoal)
 5  │   foreach Artifact artifact in artifacts do
 6  │   │   var contextSatisfaction ←
 │   │   isSatisfied(artifact.contextConditions)
 7  │   │   if contextSatisfaction then
 8  │   │   │   var plan ← new DeploymentPlan ()
 9  │   │   │   plan.add(artifact)
10  │   │   │   if artifact.dependencies == EMPTY then
11  │   │   │   │   subPlan.add(plan)
12  │   │   │   │   break
13  │   │   │   end
14  │   │   │   else
15  │   │   │   │   var depPlan ← doPlanDeployment (artifact.dependencies)
16  │   │   │   │   if depPlan != NULL then
17  │   │   │   │   │   plan.add(depPlan)
18  │   │   │   │   │   subPlan.add(plan)
19  │   │   │   │   │   break
20  │   │   │   │   end
21  │   │   │   end
22  │   │   end
23  │   end
24  │   if subPlan != EMPTY then
25  │   │   resultingPlan.add(subPlan)
26  │   end
27  │   else
28  │   │   return NULL
29  │   end
30  end
31  return resultingPlan
```

Algorithm 1: doPlanDeployment (List goals)

The Algorithm 1 works as follows: it receives as parameter as deployment request, which contains a list of goals. For each goal in the list, it queries the repository for artifacts that provides this goals (line 4). The repository returns a list of artifacts. For

each artifact the algorithm looks for a sub plan with this artifact (line 5-21). First, the context conditions are verified (line 6). If the context is satisfied (line 7), then a new plan is created with the artifact (line 8-9). If the list of dependencies of the artifact in empty (line 10), them the new plan is added to the sub plan (line 11). Else, if the artifact has a not empty set of dependencies, the algorithm is recursively called for this dependencies. If the results of the recursive call is not NULL (line 15), the resulting plan is added to the new plan and the plan is added to the sub plan (line 16-17). In both cases that a new plan is added to a sub plan, the look for a deployment plan that satisfy the selected goal is over and the inner for loop is broken (line 12 and 19) and them the sub plan is added to the resulting plan (line 25). Otherwise, if the context conditions evaluation (line 6) returns FALSE or the recursive call returns NULL, this artifact can not be deployed. The loops continues and others artifacts will be tried. The algorithm tries to come up with a sup plan for the selected goal with another artifact that provides this goal. If after all tries the sub plan is EMPTY (line 27), the deployment for the selected goal is not possible, and the algorithm returns NULL (line 28). Note that the algorithm will return NULL if for any of the goals in the request it is not possible to come up with a plan. Otherwise, the algorithm will return a valid plan.

It could be the case that there are more then one possible valid plan. But this algorithm will return the first one found. We let for future works the investigation of approaches to come up with the best alternative plan in case that more then one is valid.

6.1.2   Verifying a Plan

A deployment plan, is valid for a given context if: (i) for each artifact in the plan, for the current context, all context conditions hold. (ii) for each artifact, for all its dependencies, there is at least one artifact in the plan that provides it (the dependency).

A deployment plan satisfy a deployment request if it valid, and (iii) for each goal, in the deployment request, there is at least one artifact that provides this goal.

Being so, we can verify if a deployment plan satisfy a deployment request by executing the following steps, that verifies the properties (i), (ii) and (iii):

- Check if for all selected artifacts, all context conditions are met.

- Check if for all selected artifacts, the dependencies are within the deployment plan.

- Check if for all goals in the deployment request there are at least one artifacts that declare this goal and one that implements this goal.

6.1.3   Deployment Execution

The last step of the approach is the deployment execution. The deployment execution involves (i) coping the artifacts present in the deployment plan from the repository to the target environment. And (ii) binding the components present into these artifacts, creating the application architecture.

To bind components, the Dependency Injection design pattern can be used. The basic idea of the Dependency Injection, is to have a separate object, an assembler, that wires together client and server components at runtime[22]. The client refers to a component that uses another component (a service) through an interface. The assembler looks for an

available service (implementation of the the interface), instantiates it, and wires it into the client object.

Chapter 7

Evaluation

7.1 Evaluation

In this section we focus on the evaluation of the proposed approach. To do so we used the Goal-Question-Metric (GQM) evaluation methodology [7].

As the first step in GQM methodology we defined the following high-level evaluation goal:

Our first evaluation goal G1 is to assess the feasibility of the approach. To do so, we need to evaluate if a software architect/developer can follow the proposed patterns to refine a goal-model into components and artifacts. Also we need to evaluate if the proposed planning algorithm is capable of autonomously creating a reliable deployment plan. Such an evaluation required the definition of the following questions and metrics:

- Q1.1: For the Filling Station Advisor case study, are the goal-component-artifact patterns a feasible approach to map artifacts from the CGM of the case study?

    - Map artifacts for the Filling Station Advisor case study using proposed patterns.

- Q1.2: How long would the algorithm take to come up with a deployment plan?

    - Time to produce a plan.

- Q1.3: How reliable would a plan provided by the algorithm be?

    - Percentage of correct answers.

Since the Filling Station Advisor has a limited size and does not allow for controlled factors experiments, our second goal G2 aims to provide a more comprehensible scalability evaluation of Goalp. So we defined the following questions and metrics:

- Q2.1: How does the algorithm scale over the number of artifacts in the deployment plan?

    - M2.1: The time consumed to come up with a deployment plan.

- Q2.2: How does the algorithm scale over the variability level on the repository?

    – M2.2: The time consumed to come up with a deployment plan.

The experiments were conducted using a laptop computer with Intel i5-3337U, 12GB DDR3 1600MHz memory, and Linux (Kernel 3.16.0-77generic). OracleJDK(1.8.0 91-b14) was used to build and run the project. All experiments were implemented in Java and are available. [1] All experiments to evaluate the algorithm correctness and scalability were implemented as automated tests under Java's JUnit framework.

7.1.1 Feasibility Assessment

We validated the feasibility of the approach applying it to the Filling Station Advisor. *Question 1.1, mapping components and artifacts*

We applied the patterns described in Table 5.1 to the CGM depicted in Figure 4.2. Then we defined the artifacts that would package that components following the proposed deployment architecture style (5.1.2). With this, we mapped 21 artifacts. We concluded that the goal-component-artifact patterns is a feasible approach to map artifacts.

Question 1.2 and 1.3

We instantiated an artifact repository with the mapped artifacts. We defined 7 deployment scenarios with different contexts. The scenarios that we used where: (s1) simple phone with ODB2, (s2) smartphone with ODB2, (s3) smartphone without car connection, (s4) dash computer with GPS and no nav sys integration and (s5) dash computer, connected, with GPS and navigation system integration. Scenarios (s6) dash computer without GPS, and (s7) nav system without Internet connection or storage.
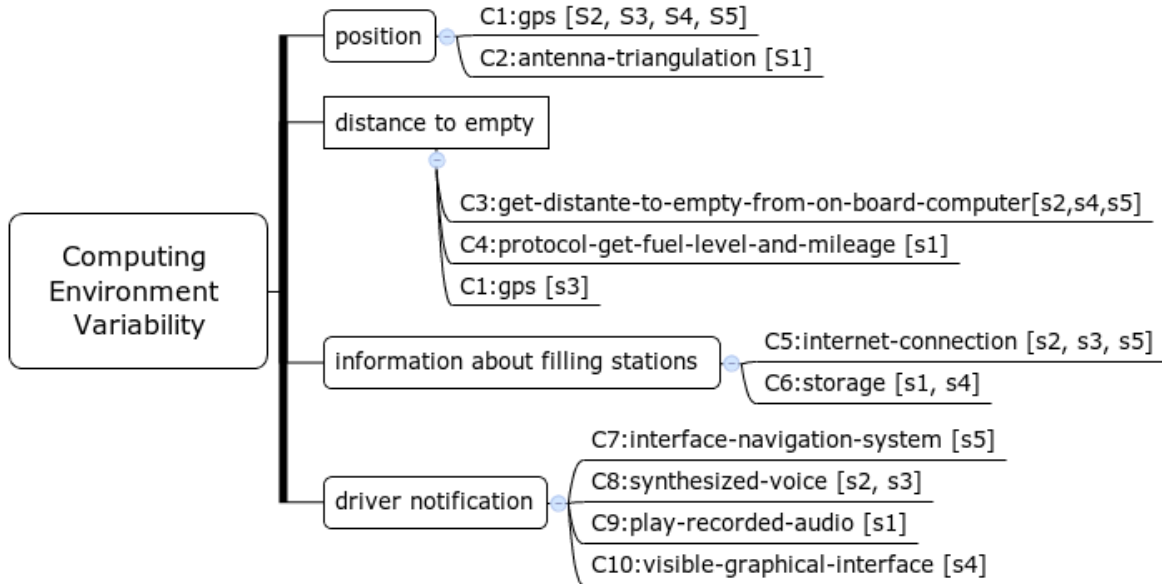

Figure 7.1: Computing Environment Evaluation Scenarios

---

[1]The evaluation experiments and the complete result set are available at `https://github.com/lesunb/goalp` Accessed on November 5th, 2016

Question 1.2: How reliable would a plan provided by the algorithm be?: Test cases were created for each scenario (s1-s7). To validate the algorithm's correctness, we verified the generated plans in each test case, asserting if the expected artifacts are in the resulting plan. For scenarios s1-s5, the planning resulted in valid plans, with the correct artifacts. For scenarios s6 and s7, the algorithm returned NULL, as there is no possible deployment plan for these scenarios. All the tests passed.

Question 1.3: How long would the algorithm take to come up with a deployment plan?: In each scenario, the time spent by the algorithm was measured. Table 7.1 shows the scenarios, the context and time spent for planning in each scenario. In the worst case, it took 9ms to come up with the plan.

Table 7.1: Time to come up with a plan

Ref.	Context	Time
s1	C2, C4, C6, C9	9ms
s2	C1, C3, C5, C8	3ms
s3	C1, C5, C8	1ms
s4	C1, C3, C6, C10	1ms
s5	C1, C3, C5, C7	1ms
s6	C3, C6, C8	1ms
s7	C1, C3, C7	1ms

7.1.2 Scalability Assessment

To evaluate the algorithm's scalability, we developed other test cases. A repository with randomly generated artifacts was instantiated. And deployment requests that generate plans with different number of artifacts were made. With this we could evaluate the impact of the generated plan size in the the planning time. The generated repository had 143,500 artifacts.

Q2.1: How does the algorithm scale over the number of artifacts in the deployment plan? We executed 100 deployment planning requests, with different levels of complexity, where the generated plans were composed of artifacts summing from 40 to 3,100 artifacts.

The observed time in function on the number of artifacts in the plan is shown if figure 7.2.

Q2.2: How does the algorithm scale over the variability level on the repository? We repeated the experiment for different levels of variability in the repository. The level meaning the number of artifacts in the repository that provides the same goals, but with different context conditions. The experiments were executed for a variability level varing from 1 to 10. The result is depicted in figure 7.3. Each curve represents a different level of variability.

At the worst case, a deployment that need 3,100 artifacts, with 10 variants for each artifact, took 3.8s to be planned. Requests that required up to 1,000 artifacts could be fulfilled in less then a half-second.
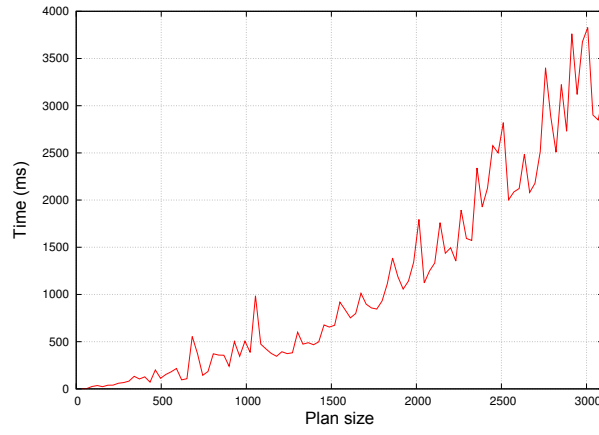
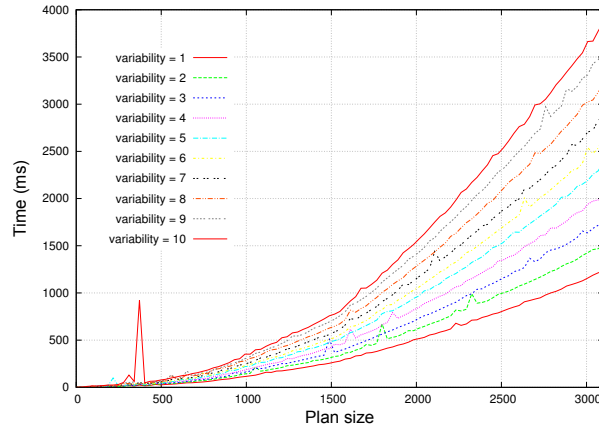Figure 7.2: Scalability over the size of plan

Figure 7.3: Scalability over variability level

7.1.3 Discussion of Results

In conclusion, the time spent planning the deployment is expect to be negligible in face the time that would take to copy the artifacts from a repository to the target environment.

Chapter 8

Conclusion

8.1   Conclusion and future work

In this paper we presented Goalp, a novel approach to tackle deployment in highly heterogeneous computing environments. Goalp allows systems deployment to heterogeneous environments, partially unknown at design-time, without requiring a system administrator. Goalp consists in support to design a system with the needed variability to handle the heterogeneity, from requirements, through architecture, and deployment. And in online support for solve the variability at deployment time, finding the correct set of artifacts that allows the user achieve its goals in a given target computing environment. Goalp uses a CGM to specify variability at requirements. Further, patterns are used to map components from the CGM and keep the variability are architecture level, and deployment level. The novelty of our approach is that we provide a systematic way to design a system with focus in variability form requirements to deployment.

Following our approach the system implemented reflects the goal-model, keeping the goals traceable to components and artifacts. Via such traceability the adequate set of artifacts is autonomously chosen achieving the target software goal in a given computing environment. Since goal-models are highly abstract models, using it to drive the system adaptation, we expect to achieve a higher level of flexibility transcending the lower-level abstraction computing layers. In addition, by using context-goal models, we can handle computing resources variability. By using CGM for deployment, rework is avoided, as CGM is a model already developed in the requirements elicitation stage.

In a preliminary evaluation, we applied the Goalp approach in a case study. Further, we evaluated the scalability of the algorithm when planning in a large scenario, using a randomly generated repository and deployment requests. The results shows that the algorithm is capable of come up with a plan, in a reasonably large scenario in seconds.

This work fits in our long term vision of a method for design systems with variability at all stages of system design, from requirements to deployment. And a self-adaptable platform that can adapts the software deployment in order to make high-level user goals achievable. This work fits in this vision by providing the knowledge and planning part in a MAPE-K[27] architecture. For future work, we plan to: (1) extend Goalp with deployment planning for multiple nodes by including delegation as another form of variability; (2) evolve Goalp deployment planning in a self-adaptive approach for deployment, based on MAPE-K, with addition of monitoring, analyzing, and executing capabilities; (3) evaluate

Goalp in a open adaptation scenario with multiple developers providing components to the environment; and (4) evaluate self-adaptation at deployment level as a method of fault-tolerance that adapts the system deployment in response to failures in resources.

Referências

[1] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, 15(4):439–458, July 2010. 3, 9, 22, 26

[2] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Requirements-driven Deployment. *Softw. Syst. Model.*, 13(1):433–456, February 2014. 15, 16

[3] Jesper Andersson, Luciano Baresi, Nelly Bencomo, Rogério de Lemos, Alessandra Gorla, Paola Inverardi, and Thomas Vogel. Software Engineering Processes for Self-Adaptive Systems. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, number 7475 in Lecture Notes in Computer Science, pages 51–75. Springer Berlin Heidelberg, 2013. iv, 7, 20

[4] Konstantinos Angelopoulos, Vítor Souza, and John Mylopoulos. Capturing Variability in Adaptation Spaces: A Three-Peaks Approach. volume 9381 of *Lecture Notes in Computer Science*, Cham, 2015. Springer International Publishing. 2, 15, 16, 20

[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Comput. Netw.*, 54(15):2787–2805, October 2010. 1

[6] Soon K. Bang, Sam Chung, Young Choh, and Marc Dupuis. A Grounded Theory Analysis of Modern Web Applications: Knowledge, Skills, and Abilities for DevOps. In *Proceedings of the 2Nd Annual Conference on Research in Information Technology*, RIIT '13, pages 61–62, New York, NY, USA, 2013. ACM. 10

[7] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley, 1994. 32

[8] Genevieve Bell and Paul Dourish. Yesterday's Tomorrows: Notes on Ubiquitous Computing's Dominant Vision. *Personal Ubiquitous Comput.*, 11(2):133–143, January 2007. 1

[9] N. Bencomo, S. Hallsteinsen, and E. Santana de Almeida. A View of the Dynamic Software Product Line Landscape. *Computer*, 45(10):36–41, October 2012. 8

[10] Nelly Bencomo, Peter Sawyer, Gordon S. Blair, and Paul Grace. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. In *SPLC (2)*, pages 23–32, 2008. 10, 15

[11] Nelly Bencomo, Jon Whittle, Pete Sawyer, Anthony Finkelstein, and Emmanuel Letier. Requirements reflection: requirements as runtime entities. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 199–202. ACM, 2010. 6

[12] Alexander Borgida, Fabiano Dalpiaz, Jennifer Horkoff, and John Mylopoulos. Requirements Models for Design- and Runtime: A Position Paper. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, MiSE '13, pages 62–68, Piscataway, NJ, USA, 2013. IEEE Press. 9

[13] Lars Braubach and Alexander Pokahr. Developing distributed systems with active components and Jadex. *Scalable Computing: Practice and Experience*, 13(2):100–120, 2012. 8

[14] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, May 2004. 3, 8

[15] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner, André van der Hoek, and Alexander L. Wolf. A Characterization Framework for Software Deployment Technologies. Technical report, 1998. 1, 9

[16] Ivica Crnkovic, Judith Stafford, and Clemens Szyperski. Software Components beyond Programming: From Routines to Services. *IEEE Software*, 28(3):22–26, 2011. 11

[17] F. Dalpiaz, A. Borgida, J. Horkoff, and J. Mylopoulos. Runtime goal models: Keynote. In *2013 IEEE Seventh International Conference on Research Challenges in Information Science (RCIS)*, pages 1–11, May 2013. 9

[18] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed Requirements Acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, April 1993. 3

[19] João Ferreira, João Leitão, and Luís Rodrigues. A-OSGi: A Framework to Support the Construction of Autonomic OSGi-based Applications. *Int. J. Auton. Adapt. Commun. Syst.*, 5(3):292–310, July 2012. 12

[20] Alessandro Ferreira Leite. *A user centered and autonomic multi-cloud architecture for high performance computing applications*. PhD thesis, Paris 11, 2014. 10, 15, 16

[21] Anthony Finkelstein and Andrea Savigni. A framework for requirements engineering for context-aware services. 2001. iv, 5

[22] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. 13, 30

[23] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004. 12

[24] Ozan Gunalp, Clement Escoffier, and Philippe Lalanda. Rondo A Tool Suite for Continuous Deployment in Dynamic Environments. pages 720–727. IEEE, June 2015. 10, 15, 16

[25] George T. Heineman and William T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 11

[26] Arvinder Kaur and Kulvinder Singh Mann. Component Based Software Engineering. *International Journal of Computer Applications*, 2(1):105–108, May 2010. Published By Foundation of Computer Science. 11

[27] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. 6, 7, 36

[28] C. Klein, R. Schmid, C. Leuxner, W. Sitou, and B. Spanfelner. A Survey of Context Adaptation in Autonomic Computing. In *Autonomic and Autonomous Systems, 2008. ICAS 2008. Fourth International Conference on*, pages 106–111, March 2008. 6

[29] Thomas Kleinberger, Martin Becker, Eric Ras, Andreas Holzinger, and Paul Müller. Ambient Intelligence in Assisted Living: Enable Elderly People to Handle Future Interfaces. In *Proceedings of the 4th International Conference on Universal Access in Human-computer Interaction: Ambient Interaction*, UAHCI'07, pages 103–112, Berlin, Heidelberg, 2007. Springer-Verlag. 1

[30] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering, 2007. FOSE'07*, pages 259–268. IEEE, 2007. 2, 12

[31] Robbert Laddaga. Self Adaptive Software SOL BAA 98 12. Technical report, 1997. 6

[32] Pattie Maes. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM, 1987. 6

[33] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93, 2000. 11

[34] Danilo Mendonça. Dependability Verification for Contextual/Runtime Goal Modelling, 2015. 9

[35] Rabeb Mizouni, Mohammad Abu Matar, Zaid Al Mahmoud, Salwa Alzahmi, and Aziz Salah. A framework for context-aware self-adaptive mobile applications SPL. *Expert Systems with Applications*, 41(16):7549–7564, November 2014. 15, 16

[36] Mirko Morandini, Fabiano Dalpiaz, Cu Duy Nguyen, and Alberto Siena. The Tropos Software Engineering Methodology. In Massimo Cossentino, Vincent Hilaire, Ambra Molesini, and Valeria Seidita, editors, *Handbook on Agent-Oriented Design Processes*, pages 463–490. Springer Berlin Heidelberg, 2014. 8

[37] Mirko Morandini, Frédéric Migeon, Marie-Pierre Gleizes, Christine Maurel, Loris Penserini, and Anna Perini. A Goal-Oriented Approach for Modelling Self-organising MAS. In Huib Aldewereld, Virginia Dignum, and Gauthier Picard, editors, *Engineering Societies in the Agents World X*, number 5881 in Lecture Notes in Computer Science, pages 33–48. Springer Berlin Heidelberg, November 2009. 8

[38] Mirko Morandini, Loris Penserini, and Anna Perini. Towards Goal-oriented Development of Self-adaptive Systems. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 9–16, New York, NY, USA, 2008. ACM. 2

[39] Loris Penserini, Anna Perini, Angelo Susi, Mirko Morandini, and John Mylopoulos. A Design Framework for Generating BDI-agents from Goal Models. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '07, pages 149:1–149:3, New York, NY, USA, 2007. ACM. 2

[40] João Pimentel, Márcia Lucena, Jaelson Castro, Carla Silva, Emanuel Santos, and Fernanda Alencar. Deriving software architectural models from requirements models for adaptive systems: the STREAM-A approach. *Requirements Engineering*, 17(4):259–281, November 2012. 2, 12

[41] Felipe Pontes Guimaraes, Genaina Nunes Rodrigues, Daniel Macedo Batista, and Raian Ali. Pragmatic Requirements for Adaptive Systems: A Goal-Driven Modeling and Analysis Approach. pages 50–64. Springer International Publishing, Cham, 2015. 9

[42] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. pages 164–182. Springer-Verlag, Berlin, Heidelberg, 2009. 12

[43] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009. 6

[44] Mazeiar Salehie and Ladan Tahvildari. Towards a Goal-driven Approach to Action Selection in Self-adaptive Software. *Softw. Pract. Exper.*, 42(2):211–233, February 2012. 9

[45] Stephen D. Smaldone. *Improving the Performance, Availability, and Security of Data Access for Opportunistic Mobile Computing.* PhD thesis, Rutgers University, New Brunswick, NJ, USA, 2011. AAI3474990. 1

[46] D. Spinellis. Package Management Systems. *Software, IEEE*, 29(2):84–86, March 2012. 10

[47] Daniel Sykes, Jeff Magee, and Jeff Kramer. FlashMob: Distributed Adaptive Self-assembly. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 100–109, New York, NY, USA, 2011. ACM. 12

[48] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. 11

[49] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.1.* 2007. 12

[50] Axel Van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures*, pages 25–43. Springer, 2003. 2, 12

[51] Michael Woolridge and Michael J. Wooldridge. *Introduction to Multiagent Systems.* John Wiley & Sons, Inc., New York, NY, USA, 2001. 8

[52] Eric Siu-Kwong Yu. *Modelling Strategic Relationships for Process Reengineering.* PhD thesis, University of Toronto, Toronto, Ont., Canada, Canada, 1996. UMI Order No. GAXNN-02887 (Canadian dissertation). 3, 8

[53] Yijun Yu, Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Julio CSP Leite. From goals to high-variability software design. In *Foundations of Intelligent Systems*, pages 1–16. Springer, 2008. 2, 9, 12, 20, 22