



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.<sup>a</sup> Dr.<sup>a</sup> Genaina Nunes Rodrigues

Brasília  
2016

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

Coordenadora: Prof.<sup>a</sup> Dr.<sup>a</sup> Célia Ghedini Ralha

Banca examinadora composta por:

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

### **CIP — Catalogação Internacional na Publicação**

Rodrigues, Gabriel Siqueira.

Autonomic Goal-Driven Deployment in Heterogeneous Computing Environments / Gabriel Siqueira Rodrigues. Brasília : UnB, 2016.

99 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2016.

1. dependabilidade

CDU 004.4

Endereço: Universidade de Brasília  
Campus Universitário Darcy Ribeiro — Asa Norte  
CEP 70910-900  
Brasília-DF — Brasil



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.<sup>a</sup> Dr.<sup>a</sup> Genaina Nunes Rodrigues (Orientadora)  
CIC/UnB

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

Prof.<sup>a</sup> Dr.<sup>a</sup> 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, like Internet of Things (IoT). Such applications are intended to execute in a broad range of devices with different available computing resources. In order to handle some kind of heterogeneity, 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. However, such approaches are not scalable to highly heterogeneous environments. In highly dynamic and heterogeneous environment it is hard to predict the computing environment at design-time, implying likely undecidability on the correct configuration for each environment at design-time. In our work, we propose Goalp: a method that allows autonomous deployment of systems by reflecting about the goals of the system and its computing environment. By autonomous deployment, we mean that the system can find the correct set of components, for the target computing environment, without human intervention.

We evaluate our approach on the filling station advisor case study where an application advises a driver where to refuel/recharge its vehicle. We design the application with variability at requirements, architecture, and deployment, which can allow the designed application be executed in different devices. For scenarios with different environments, it was possible to plan the deployment autonomously. Additionally, the scalability of the algorithm that plan the deployment was evaluated in a simulated environment. Results show that using the approach it is possible to autonomously plan the deployment of a system with thousands of components in few seconds.

**Keywords:** dynamic deployment, heterogeneous computing environment, context- and goal-oriented requirements engineering

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Definition . . . . .	1
1.2	Proposed Solution . . . . .	3
1.3	Contributions Summary . . . . .	3
1.4	Structure . . . . .	4
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Context-aware Systems . . . . .	5
2.2	Self-Adaptive Systems . . . . .	6
2.2.1	Development of Self-Adaptive Systems . . . . .	7
2.3	Goal Modeling . . . . .	8
2.3.1	Software Deployment . . . . .	9
2.4	Software Components . . . . .	10
2.4.1	Component-Based Adaptation . . . . .	11
2.4.2	From Goals to Components . . . . .	12
<b>3</b>	<b>Filling Station Advisor</b>	<b>14</b>
3.1	Motivating Example: The Filling Station Advisor . . . . .	14
<b>4</b>	<b>The Goalp Approach</b>	<b>17</b>
4.1	Offline . . . . .	18
4.1.1	Goal Modelling . . . . .	18
4.1.2	Component Mapping . . . . .	19
4.1.3	Packaging . . . . .	21
4.1.4	Development Process . . . . .	23
4.1.5	Activities . . . . .	23
4.2	Online . . . . .	24
4.2.1	Autonomic Deployment Planning . . . . .	25
4.2.2	Deployment Execution . . . . .	28
<b>5</b>	<b>Evaluation</b>	<b>30</b>
5.1	Feasibility Assessment . . . . .	31
5.2	Scalability Assessment . . . . .	32
5.3	Threats to validity . . . . .	34

<b>6</b>	<b>Related Work</b>	<b>36</b>
6.1	Goal Oriented Approaches . . . . .	36
6.2	Automatic Deployment Approaches . . . . .	36
6.3	Package managers . . . . .	37
<b>7</b>	<b>Conclusion</b>	<b>39</b>
7.1	Conclusion and future work . . . . .	39
	<b>Referências</b>	<b>41</b>

# List of Figures

2.1	Context-aware services framework by [24]	5
2.2	MAPE-K Reference Architecture	6
2.3	A Life-cycle model for Self-Adaptive Software System[3]	7
2.4	Artifacts Deployment	10
3.1	CGM of the filling station advisor	15
3.2	Variability in the Computing Environment	16
4.1	Overview	17
4.2	Dependency Graph	22
4.3	Roles collaboration	23
4.4	Deployment Process Activities	24
4.5	Goald Autonomic Deployment	25
4.6	The Goalp Deployment metamodel	26
4.7	Representation of OSGi bundles lifecycle	28
5.1	Computing Environment Evaluation Scenarios	31
5.2	Passing Tests	32
5.3	Scalability over the size of plan	33
5.4	Scalability over variability level	34

# 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 have connectivity capability and can form networks. These networks can be rich computing environments as each device brings different computing resources. This presents a great potential, but developing software that harvests the capability of such environment is 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 [11], Internet of Things (IoT)[8], Assisted Living[35] and Opportunistic Computing[52] 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[17]. It involves planning which artifacts should be deployed, moving compatible artifacts to the target environment, configuring the environment and starting execution. *Deployment planning* is a specially challenging activity, it requires analyzing the environment and the software architecture to solve variabilities, and coming up with which software artifacts should be present in the deployment.

### 1.1 Problem Definition

Current software deployment approaches do not suit highly heterogeneous computing environment[41]. The simplest approach to deployment as a whole is manual configuration, in which a human conducts all steps in the deployment planning and execution. It is normally applied when developing customized software that will be executed in devices managed by the development team. Such approach does not scale for applications that target massive use, because it requires the deployment to be executed by a person with knowledge about the application internals[4]. Another approach, common in cloud environments, is the use of scripts to automate software deployment execution[53]. Such 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, it does not scale as the script relies on a centralized model created at design-time. *Software*



*store* is another alternative approach. Typically, the developer uploads to the store back-end site the software configuration for each kind of target device, solving any variability at this point. In such cases, the deployment execution can rely on actions by the end-user such as accessing the store interface, searching for the application, and initiating the installation of the application. Neither scripts nor software stores are suitable for heterogeneous environments because they are highly dependent 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 meant to run in a broad range of configurations of the computing environment.
- **Challenge 2: uncertainty at design-time.** The system architect/developer cannot precisely ascertain the configuration of the end user computing environment.
- **Challenge 3: deployment should be autonomous.** A deployment specialist is unlikely available at deployment time for a particular environment, so the deployment should be planned and executed autonomously.

Many works have investigated the relation of goals and architecture of a system [36][45][46][47][59]. Some works in the literature have investigated variability in goal models with adaptation purpose [5][61]. These works show that goal modeling is a promising approach to manage variability at the design of the software. But, to the best of our knowledge, none investigated goal models at deployment level. Accordingly, our first research question emerges:

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

With RQ1 we are interested in extending goal-oriented variability models to deployment level. By addressing RQ1, we expect to allow the deployment of the system to 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 of user goals in the target computing environment.

## 1.2 Proposed Solution

This work proposes Goalp: a method that follows a goal-oriented approach for deployment in highly heterogeneous computing environments, capable of determining a suitable configuration from a general set of configurations for deployment. In particular, we focus on autonomous deployment planning as the major part of the deployment in heterogeneous environments. In our approach, the planning is executed autonomously, that is, it does not require a human 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 alternative strategies), and (iii) the *restrictions* to the strategies (i.e., the resources needed). Part (i) comprehends requirements modeling. Part (ii) comprehends artifacts containing software components and metadata. Part (iii) comprehends 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 wants to achieve, where system requirements are modeled as intentions of actors in strategic goals[16][21][60]. 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 models into software components; (ii) a description on how to create artifacts as packaged components with deployment metadata information; (iii) a deployment metamodel that characterize deployment information; (iv) an algorithm to analyze the deployment metamodel and, for a given computing environment together with a set of goals, select an appropriate set of artifacts that allows the achievement of the goals in the computing environment. Goalp was evaluated in a case study and using a randomly generated workload. The 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 major contributions of this proposal.

1. A method to develop systems for heterogeneous computing environments that supports variability for software deployment, comprising:
  - 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 comprising:
  - A metamodel that describes the deployment
  - An algorithm to autonomously planning the deployment
  - A Java implementation of the algorithm

## 1.4 Structure

This dissertation is organized as follows. Chapter 2 introduces the theoretical background underlying our work. Chapter 3 Presents the case study of the Filling Station Advisor. Chapter 4 presents patterns and guidelines to develop software to heterogeneous computing environments and the support for autonomous deployment. Chapter 5 depicts the evaluation of Goalp. Chapter 6 presents most relevant related literature work and Chapter 7 concludes 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 those able to adapt their behavior according to changing circumstances without user intervention. Finkelstein and Savigni [24] describe a framework for context-aware services. Their approach is depicted in Figure 2.1.



Figure 2.1: Context-aware services framework by [24]

*Environment* is whatever in the world provides a surrounding in which the agent is supposed to operate. The environment comprises such things as characteristics of the device that the agent is expected 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 watch relevant environment properties and keep a runtime model that represents those properties. By reasoning about that model the system can change its behavior. A *context* can be either an *activator* of goals or a *precondition* on the applicability of a certain strategy to reach a *goal*.

A *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

concrete and 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 modeled element exists if one of them changes, this leads to a corresponding effect upon the other [38]. 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 the requirements reflection vision [14], a system should keep software requirements model at runtime, and use such model to drive the system adaptation.

## 2.2 Self-Adaptive Systems

Self-adaptiveness 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."*[37]. 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[50].

A key concept in self-adaptive systems is the awareness of the system. It has two aspects[50]:

- *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.[34] 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”.

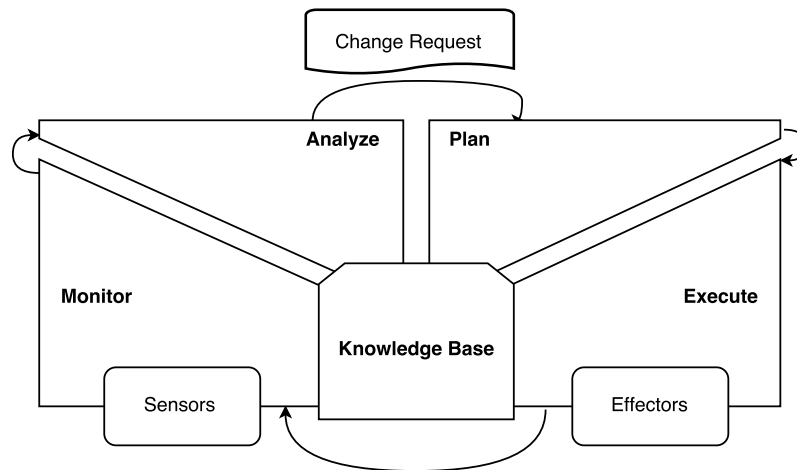


Figure 2.2: MAPE-K Reference Architecture

MAPE-K is a reference architecture originally proposed for autonomic computing [33] and that is often used as a model for architectures of SAS. It has a control loop, realized by

a simple sequence of four activities: *monitor*, *analyze*, *plan*, *execute* and a *knowledge* are. The adaptive system interacts with the environment or managed sub-system through *sensors* and *actuators*. The *monitor* activity collects data from *sensors*. That data is *analyzed*, and if a need of change is identified, a change request is dispatched, then an adaptation should be *planned*. The resulting plan is passed to the *execute* activity, which is performed through *actuators*.

### 2.2.1 Development of Self-Adaptive Systems

For SAS some activities that traditionally occur at development-time are moved to runtime. Andersson et al. [3] proposed a process for development of adaptive systems. In their approach, activities performed externally to the adaptive system are referred as *off-line activities*, and activities performed internally in the adaptive system are *on-line activities*. Off-line activities are mainly related to the design of the system, while online activities are related to the run-time of the system.

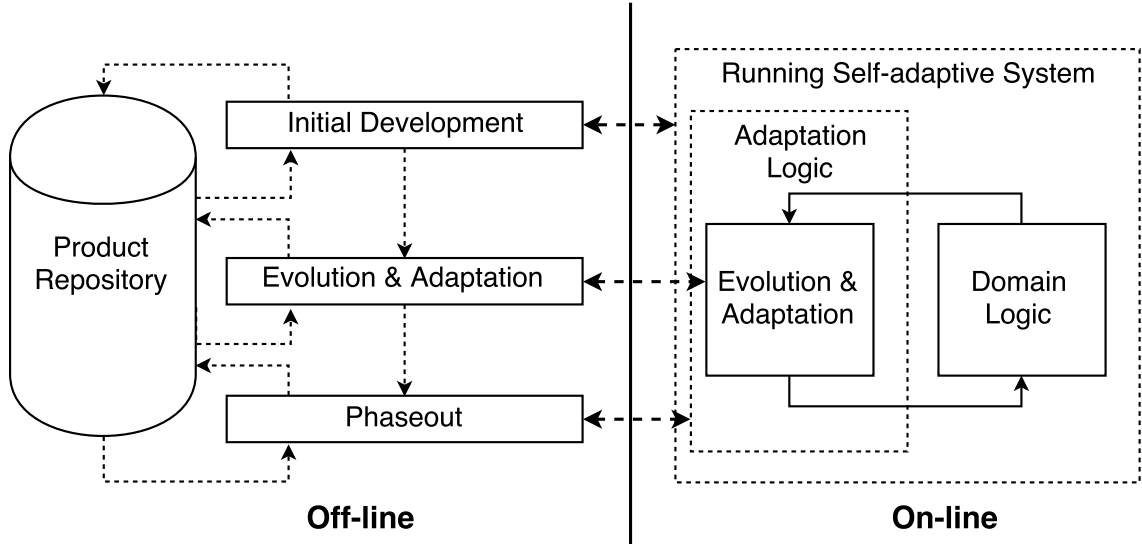


Figure 2.3: A Life-cycle model for Self-Adaptive Software System[3]

The left-hand side of Figure 2.3 represents a development life-cycle model. Off-line activities work on design model and source code in a product repository and produce the artifacts that will be used in the running system. The right-hand side of Figure 2.3 depicts a running SAS. In this approach, we have *Domain Logic* that is responsible for final user goals achievements. *Adaptation Logic* is responsible for adapting the system in response to changes in the environment. In addition, the adaptation logic implements a control loop in line with the monitor-analyze-plan-execute (MAPE) loop [33].

## 2.3 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 [44]. Goals capture the various objectives the system under consideration should achieve. In particular, Tropos[16] is a methodology for developing multi-agent systems that uses goal models for requirement analysis.

### The Tropos key concepts

Tropos uses a modeling framework based on  $i^*$  [60] which proposes the concepts of actor, goal, plan, resource and social dependency to model both the system-to-be and its organizational operating environment [16] [43].

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** is an entity that has strategic goals and intentionality

**Agent** is the physical manifestation of an actor.

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

**Plans** represent a way of doing something. Plans are concrete actions or procedures that an agent can perform. The execution of a plan can be a means for satisfying a goal or for *satisficing* (i.e. sufficiently satisfying) a soft goal.

**Resource** represents a physical or an informational entity.

**Dependency** it is a relationship between two actors that specify that one actor (the *depended*) has 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** represents both the *ability* of an actor to perform some action and the *opportunity* of doing so.

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 traditional requirements tool, as such it must capture the solution space 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 [15]. Traditional goal models can be named design-time goal model (DGM). Dalpiaz et al.[20] describe a method for extending Design-time Goal Models (DGMs) to

create Runtime Goal Models (RGM). RGMs can be used to analyze the system’s runtime behavior. Other works relate goal models with another dynamic aspects of systems, such as configuration [61], behavior [20], probability of achieving success [40] and achievability of goals [48].

Salehie et al. [51] propose a run-time goal model and its related action selection. They model 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 affect 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 makes them visible and traceable.

## 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.

Mendonça et al. [40] propose GODA: a methodology for dependability analysis by which the software engineer, at design-time, annotates the goal decomposition in goal model and specify context variables. A special tool generates a formula to evaluate for a given context the probability of achieving a goal at runtime.

### 2.3.1 Software Deployment

Software deployment refers to all activities that make a software system available for use[17]. These activities result 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 burning software into a chip. In consumers’ personal or business domain, for a desktop platform, the deployment can consist of an installation process with collaboration between a person and a script that automates some steps. In an enterprise domain, for a web platform it can consist in coping and editing some files in a couple of machines. In many of those scenarios software will be periodically updated, frequently becoming unavailable during the update process. The complexity of the software deployment can also vary as a 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.

Deployment artifacts are the artifacts needed at the deployment environment. Artifacts are built at development and build environment. Built artifacts are moved to a delivery system where they can be accessed 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*[31] environment applies automation in building and getting components ready



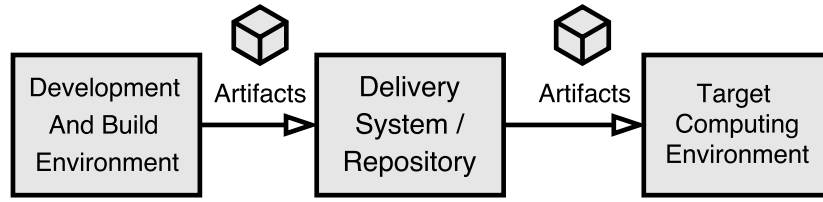


Figure 2.4: Artifacts Deployment

to delivery. In such environment if a developer pushes changes to a code repository, components are automatically built and published to delivery system. The build process commonly involves fetching build dependencies, compiling source code, running automated quality control (tests and static analysis) and packaging components into artifacts. Artifacts are published if target quality policies are met. Fundamental to continuous integration environments are *Dependency Management Systems* tools, such as Maven[7] for Java platform. These tools simplify the management of software dependencies [54]. Such 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 responding to dynamisms in a known environment. This could be costs and failures in a cloud environmet [23], changes in managed resources [28], and changes in the context of operation [13].

*Continuous delivery*[31] extends the continuous integration environment, moving components from the delivery system to a target computing environment with none or minimum human intervention.

In the industry, package managers such as aptitude/apt-get(Debian based Linux distributions) [6], yum (Red Hat based Linux distributions) [55], Homebrew (MacOS)[30] and Chocolatey (Windows)[18] are capable of solving dependencies and deploying software. They require that a managed application declare their dependencies by name and version. DevOps[9] 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*), following best practices of software development. That movement favors the documentation, reproducibility, automation and scalability. DevOps allows for management of scalable computing environments. It can offer a 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 require that highly specialized system administrators to analyze the environment and create environment configuration descriptors.

## 2.4 Software Components

Heineman defines *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”[29].

Software components are units of composition. Software systems are built by composing different components. Software components must conform to a component model by having contractually specified interfaces and explicit context dependencies only.[57].

A *component interface* “defines a set of component functional properties, that is, a set of actions understood by both the interface provider (the component) and user (other components, or other software that interacts with the provider)”[19]. 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.[29] 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[19]. *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[32]. The architectural view of a software can be formalized via an architecture description language (ADL)[39].

Component-based software engineering (CBSE) approach consists of building systems from components as reusable units and keeping component development separate from system development[19].

CBSE is built on the following four principles[19]:

- *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 extend the system’s functionality.
- *Composability*. A system should support the composition of functional properties (component binding). Composition of extra functional properties, for example, composition of components’ reliability, is another possible form of composition.

### 2.4.1 Component-Based Adaptation

In the literature, there has been proposals of framework for architecture and components based adaptation.

Rainbow[27] is a framework for architecture based self-adaptation. It keeps a model of the architecture of the system and can be extended with rules to analyze the system behavior at runtime, find adaptation strategies and perform changes. It separates the functional code (internal mechanisms) from adaptation code (external mechanism) in a schema called external control, influenced by control theory.

MUSIC[49] project provides a component-based middleware for adaptation that proposes to separate the self-adaptation from business logic and delegate adaptation logic to generic middleware. 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 [56] 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 [56]. Flashmod uses a three-layer model: goals, management and components proposed by Kramer and Magee [36], extending it to allow distributed agreement in a given configuration.

OSGi[58] is a Java centric platform that allows dynamic bind and unbind of components, usually named bundles. Ferreira et al.[22] proposed a framework for adaptation based on OSGi.

## 2.4.2 From Goals to Components

Lamsweerde [59] presents a method for deriving architecture from KAOS goal model[21]. Firstly, an abstract draft is generated from functional goals. Secondly, the architecture is refined to meet non-functional requirements such as cohesion.

Pimentel et al. [47] present a method using i\* models to produce architectural models in Acme, a language employed to describe architectural models. Firstly, i\* model is transformed into a modular i\* model employing a horizontal transformation. Secondly, an architecture model is created from the i\* modularized model employing a vertical transformation. Architectural design models is made easier by the presence of actor and dependency concepts.

Yu et al. [61] proposed an approach for keeping the variability that exists in the goal model into the architecture. It presents 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 directly derived from the goal name. Goals refinements result in the 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;
}
```

## Dependency Injection

Dependency Injection is a pattern that allows for wiring together software components that were developed without the knowledge about each other. [26]

In OO languages normally one instantiates an object from a class using an operator (*new* for Java) and a reference to such class. Interfaces create architecture independence. Yet, even using interfaces we can have static dependencies at some point, at the implementation instantiation. The object that is instantiating (the client) is dependent on the referenced class (the service).

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

In case of strongly typed languages, normally one will get an exception if the referenced class is not present.

The basic idea of the Dependency Injection is to have a separate object, an *assembler*, that wires together the components at runtime[26]. The client class refers to the service using its interface (the service interface). The assembler can use alternative ways to the *new* to instantiate an object so that the wiring between client objects and implementation service classes could be postponed to runtime. Using reflexive platforms we can eliminate the static dependency as the platform can find available interfaces implementations at runtime. The assembler can use reflexive capabilities of the platform to discover the available implementations and instantiate them.

In the context of component-based adaptation, decoupling client components from service components would be specially useful, allowing runtime reasoning about what implementation to choose.

# Chapter 3

## Filling Station Advisor

### 3.1 Motivating Example: The Filling Station Advisor

In this work, we use a filling station advisor application as a case study to exemplify the application of our approach. Filling station here refers to a place where the car can be refueled or recharged (gas station/petrol or charge station). The main goal of the filling station advisor is to give directions to a 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 (s1) 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 (s2), in which the application is running in onboard computer of an Internet connected self-driving car, we could use a more precise distance-to-empty data from onboard 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 main goal of the application is refined in the following five goals, each one with its own computing resources requirements:

**Get 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

onboard 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.

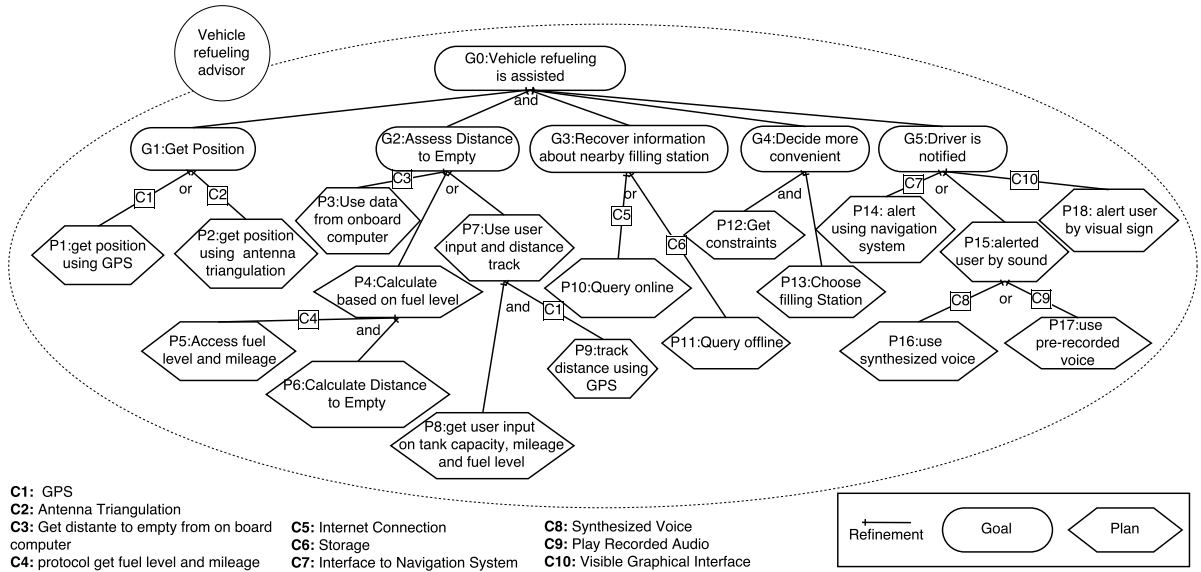


Figure 3.1: CGM of the filling station advisor

The CGM presented in Figure 3.1 depicts the goals to be achieved by the Filling Station Advisor. The root objective *G0: Vehicle refueling is assisted* is AND-refined into 5 others objectives G1, G2, G3, G4 and G5. In the Goal modeling semantics it means that in order to achieve the root Goal G0, the agent should achieve the goals G1, G2, G3, G4 and G5. *G1: Get Position* has a means-end association with P1, P2 and P3. It means that the goal G1 can be achieved by executing that plans. As it is an OR-refinements, it means that G1 can be achieved by successfully executing any of the plans P1, P2 or P3.

This OR-refinement introduces a variability to the system, allowing it to achieve to root goals in different ways. The contexts C1 in the association between P1 and G1 means that the Plan P1 is executable if the context C1 holds. Context conditions on the example are of the type "required context" [1]. These annotations means that a certain way for achieving (executing) a goal (plan) is applicable if the condition holds for the context.

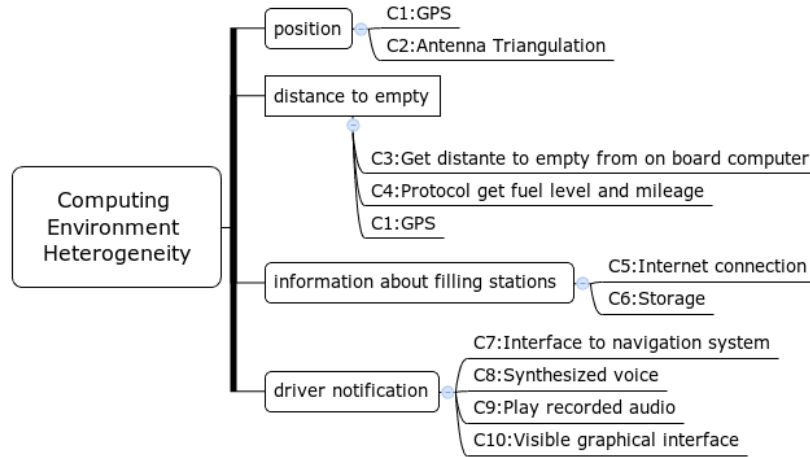


Figure 3.2: Variability in the Computing Environment

Figure 3.2 outlines the context space of the target computing environment. It contains variability contexts that are expected to occur in 4 subgoals (G1-G3 and G5) of the application.

# Chapter 4

## The Goalp Approach

Following the model proposed 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 the target environment and result in the deployment of the system. Figure 4.1 presents an overview of the process.

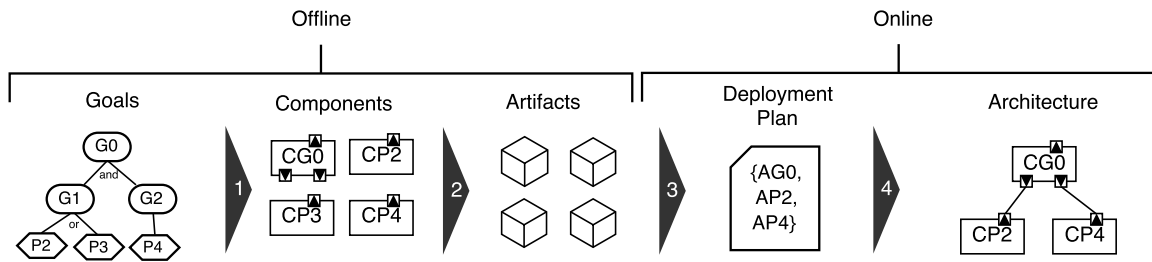


Figure 4.1: Overview

Figure 4.1 depicts an overview of the approach, which consists in four major activities: (1) goal modeling, (2) context goal modeling, (3) component analysis and (4) component development.

In the offline activities, occurs the design of the application. In the first step of our approach, components are mapped from the system CGM using patterns. In the second, these components are packaged into artifacts together with their metadata, which describes what goals the artifact provides, its context conditions and dependencies.

In the online part of the approach, the Goalp deployment is autonomously planned in the target computing environment. The deployment planning is responsible for looking 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.

We further explain such activities in details as follows.



## 4.1 Offline

Previously, goal-driven approaches were proposed for introducing variability at requirements, context modeling, software behavior, and software architecture[5][61]. 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 monolithic software, we implement in the same codebase all functionalities, then all code is built and deployed together. In the Filling Station Advisor example, if implementing it as a monolithic 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 on specific resources in the environment. By encapsulating dependencies of specific resources into components, 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. Such variability is useful as it allows the deployment of components only to environment that have the required resources.

Regarding the Filling Station Advisor example, depicted in Figure 3.1, for goal *G1* (Get Position), components can be implemented providing the actual position of the device by means of GPS or antenna triangulation. Such components can be packaged into different artifacts that will only be deployed when the target environment has the appropriate resources.

### 4.1.1 Goal Modelling

A systematic way of analyzing 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.

**Definition 1 (Resource)** *A resource is a specific computing capability that could be available in the computing environment and used in plans. A resource receives a label.*

Regarding the Filling Station Advisor, examples of relevant resources are *GPS*, (labeled *c1*) and *antenna triangulation* (labeled *c2*). In order to reason about available resources in the target computing environment, a context model is used.

In Goalp, the context of a target computing environment is reified as a set of resources. The semantics is that if the resource is present in the context, the associated computing capability is available in the target computing environment. In our example, the set [*c1*, *c5*, *c8*] is an example of context, representing that GPS, connection to the Internet and voice synthesizing are capabilities available in the computing environment.

Plans can require resources in order to be applicable, for example, *P1: get position using GPS* requires the resource GPS to be available. The Deployment Goal Model (DGM) extends a goal model with resource related restrictions to the applicability of plans.

**Definition 2 (Deployment Goal Model)** *A Deployment Goal Model (DGM) is a tuple  $(M, ctx\_spc, ctx\_cond)$  where:*

- *M is a design-time goal model[20] defined as a tuple  $(N, R)$ , where N is a set of goals and plans in the model, and R the corresponding set of relationship links between the elements in N.*
- *ctx\_spc is a set of resources in the model.*
- *ctx\_cond:  $R \rightarrow ctx\_spc$  associates a relationships in R with resources in ctx\_spc.*

A context condition is satisfied if its associated resource is present in the context. For example, in a scenario with context  $ctx=[c1, c5]$ , the context condition c1 is satisfied. *Context conditions* are restrictions to the applicability of a plan and are used to solve variability at deployment. In Figure 3.1, the goal *G1: Get position* has two alternatives to be achieved: by executing the plans *P1: get position using GPS* or *P2: get position using antenna triangulation*. The plan P1 is applicable if the context condition c1 (GPS) is satisfied, which is the case when GPS capability is available in the target environment.

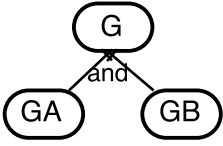
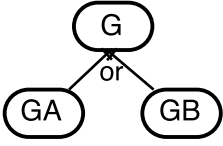
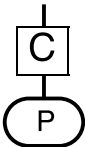
### 4.1.2 Component Mapping

Components are architectural units. In our proposal, components definitions are mapped from the DGM and them developed by the architect/developer. We postulate that components should *provide* goals which means that they should be executed to *fulfill* a goal at runtime.

The patterns present in Table 4.1 are used to map components based on the DGM of the system. By mapping components we mean identifying which component should be developed in order to reflect the DGM of the system. By using the proposed patterns, the variability present in the DGM is kept at the architecture of the system. Theses patterns are an extension of Yu et al.[61] patterns for the Goals-Component view. We extended Yu et al.[61] patterns with context conditions. The presented patterns are described using goals but they can be applied for goals and plans without distinction.

And-refinements result in components that define a strategy to achieve a given goal by achieving two or more sub, more concrete, goals. Mapping components from an And-refinement results in: (i) a root interface that describe what component provides, (ii) interfaces for each sub-goal, (iii) a component that provides the root interface and requires each interface generated for sub goals. The latter component implements a strategy to achieve its provided goal. It coordinates the sub more specific goals by calling them and passing one result as the input of another, when applicable. As an example, applying And-refinement patterns for Root Goal - *G0: Vehicle refueling is assisted* - 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.

Table 4.1: Contextual Goal Model to components - patterns

And-Refinement	 <pre> graph BT     G((G))     GA((GA))     GB((GB))     GA -- and --&gt; G     GB -- and --&gt; G </pre>	<pre> Component CG {   provides IG;   requires     IGA, IGB; } </pre>
Or-Refinement	 <pre> graph BT     G((G))     GA((GA))     GB((GB))     GA -- or --&gt; G     GB -- or --&gt; G </pre>	<pre> Component CG {   provides IGA; } Component CG {   provides IGB; } </pre>
Context-condition	 <pre> graph BT     G((G))     P((P))     C[C]     G -- C --&gt; P </pre>	<pre> Component CG {   provides IG;   condition C; } </pre>

```

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

```

Applying Or-refinements patterns results in a root interface definition and in multiple implementations. In the DGM, when Or-refinements are associated with context conditions, it allows for alternative strategies using different resources in the computing environment. Using the association of OR-refinement and context-conditions in the component analysis we preserve the variability present in the DGM into the architecture of the system.

For example, in the Filling Station Advisor, applying the patterns for G1:Get Position, P1:get position using GPS and P2:get position using antenna triangulation will result in the following components:

```

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

```

```

    condition C2;
}

```

The two components CP1 and CP2 provide the same goals but have different context conditions (C1:GPS and C2:antenna triangulation). It means that we can achieve the same goal using any of the two resource, by deploying one of the two component variants. That variability in the architecture allows for the adaptation to the heterogeneity in the target environment.

### 4.1.3 Packaging

*Artifacts* are deployment units. From the deployment point of view, the components and interfaces should be packaged in a file to be distributed. We name *artifact* as such file. An artifact should follow a standard packaging schema, so a package manager can manipulate it. In our approach, we propose to include into the artifact metadata to describe its related goals, context conditions, and dependencies. That metadata reflects 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's 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 come up with 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 an 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 for resources in the computing environment that are beyond the deployment agent capacity of management. Such conditions can be related to hardware implementation, e.g. presence of a GPS-module, our platform lower level software implementation, e.g. access to a protocol to query vehicle onboard computer data. If a context condition does not hold, there is nothing that can be done at deployment time to change that. *Dependencies*, on the other hand, refer to the need on other artifacts. It is specified in terms of *Provided Goals*. Like other dependency management approaches, an artifact can depend on other artifacts. Different from other approaches, we specify the dependency not as specific version of an artifact identification but in terms of what an artifact provides. In the Filling Station Advisor, the artifact A0, that packs component G0 depends on IG0-definition and IG0, the interface that describes goal G0. To satisfy the dependency, an artifact that provides IG0-definition should be available, as well as at least one artifact that provides IG0. Figure 4.2 depicts the dependency relationship between artifacts. A1 is a root interface and have 3 dependencies, that

are provided by A2, A3 and A4. A2 and A3 have one common dependency, provided by A5. A5 and A6 has no dependencies.

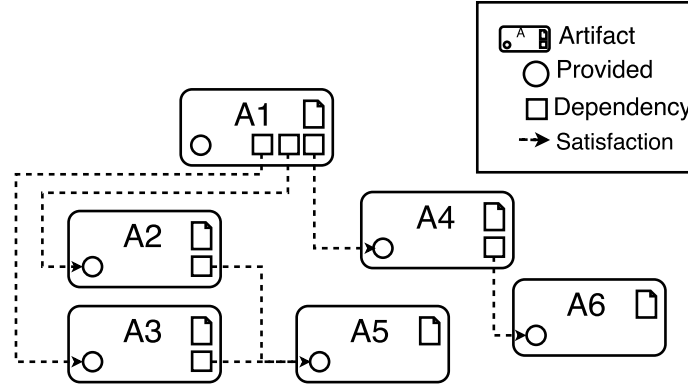


Figure 4.2: Dependency Graph

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

### Deployment Architectural Style

In order to maximize the flexibility of systems following Goalp deployment approach, we propose the following architecture style to create artifacts of three types:

**Definition** artifacts that specify the interface of goals. It contains interfaces declarations and data model. It specifies 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 does not implement plans that use specific resources in the computing environment.

**Plan implementation** that artifacts contain the domain logic implementation. The plan could have dependencies on specific resources in the computing environment. In a dependency tree, artifacts of plan implementation type are the leaves. Plan implementation artifacts provides low-level goals and can have dependencies and context conditions.

These types of artifacts are meant to increase the flexibility of deployment.

### 4.1.4 Development Process

In previous subsections we proposed techniques to support the design of software with variability from requirements to deployment. In this section we present activities to apply such techniques in a software development process.

#### Roles

The proposed process considers three roles: users, requirements engineers and software architects. Figure 4.3 summarizes the collaboration between the roles.

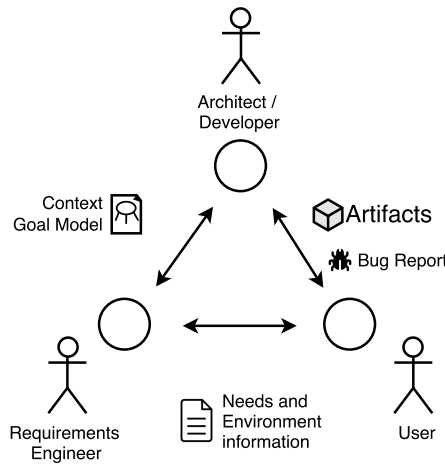


Figure 4.3: Roles collaboration

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

**Requirements Engineer** Is responsible for translating users goals to a goal model. Also is responsible for analyzing the different contexts that the system is meant to operate and how they affect the goals, defining the DGM of the system.

**Architect** Projects the software architecture so 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.

### 4.1.5 Activities

Figure 4.4 describes the development process activities.

#### Goal Modeling

This phase is coordinated by a requirements engineer with the participation of a domain specialist, possibly the user. In this activity a goal model is created. At the goal

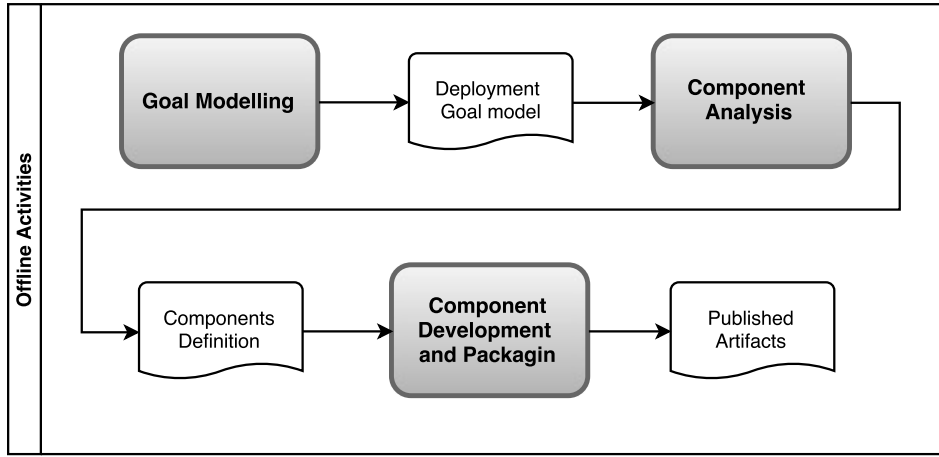


Figure 4.4: Deployment Process Activities

model it is identified the solution space, what the system should achieve, and possible strategies to achieve the goals. Also, the goal model creates a common language between users and software engineers. In this activity, relevant resources should be identified and the goal model should be annotated with *context conditions* related to the computing environment using the formalism described in Section 4.1.1.

### Component Analysis

The architect is the responsible for this activity. It receives as input a DGM. Then, variability points, components and its interfaces are identified. Component interfaces are created following the guidelines described in Section 4.1.2.

### Component Development

The architect is the responsible for this activity. Component development includes the coding, build and test of software components. Then, components are package into artifacts and put in the repository as described in Section 4.1.3.

## 4.2 Online

In the online part of the approach, the artifacts present in the repository are autonomously deployed to the target computing environment.

Figure 4.5 depicts the online activities. In the first step, a user interested in using a computing environment to achieve a set of goals submits to such environment which goals it wants to achieve in the form of a deployment request. In the second step, the target environment realizes a deployment planning by analyzing the available computing resources and artifacts present in the repository, then generating a deployment plan: a selection of artifacts that can allow for the goals achievement in the available computing environment. Finally, the deployment is executed by fetching the selected artifacts from the repositories and binding them.

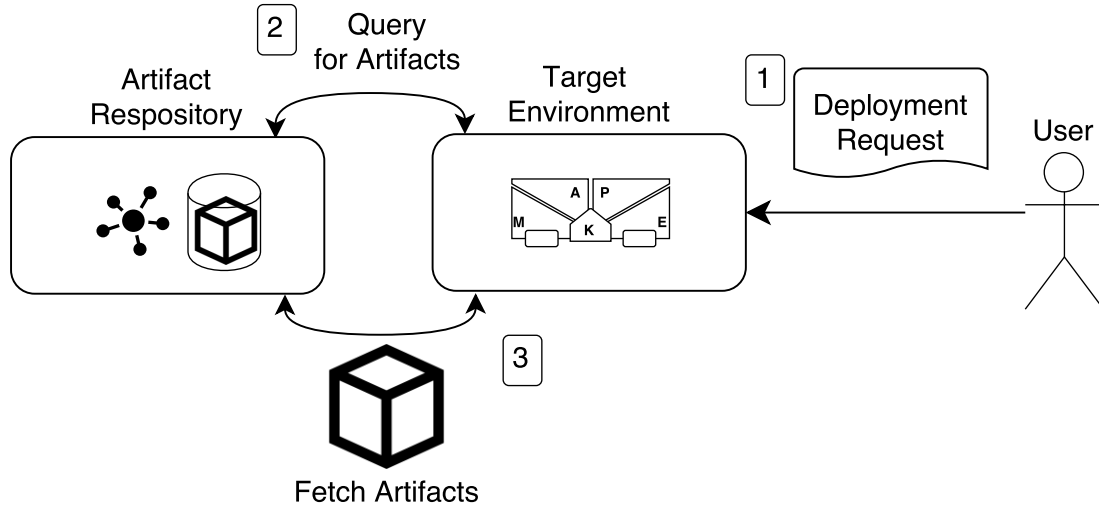


Figure 4.5: Goal-driven Autonomic Deployment

### 4.2.1 Autonomic Deployment Planning

The deployment planning is the core of the online part of the presented approach. Its objective is to solve the variability present in the design, allowing the system to autonomously adapt to the target computing environment. In this Section 4.2.1 we present the metamodel used and an algorithm to come up with a deployment plan.

#### Metamodel

The metamodel of Goalp consists of 6 major elements: *Goal*, *Artifact*, *Agent*, *Repository*, *Deployment Request*, and *Deployment Plan*. Figure 4.6 presents Goalp metamodel. *Artifact* is the central entity at deployment level. As described in the Section 4.1.3, artifacts have *provided goals*, *context conditions*, and *dependencies* which create relations of dependency between artifacts, so that an artifact that has a goal dependency is dependent on an artifact that provides such goal.

The *Agent* can accept deployment requests, an action that should trigger the deployment planning. An agent knows a *repository* where it looks for artifacts. A *repository* has a set of artifacts that can be queried by the `queryForArtifacts` method. The method `queryForArtifacts` receives a goal as the argument and return all artifacts in the repository that provide that goal. An *agent* can verify *context conditions* by `isSatisfied` method.

The *Deployment Request* is a set of goals that an external entity sends to an agent, requesting it to plan a deployment. *Agent's* `doPlanDeployment` method returns a *Deployment Plan*, which is a set of artifacts that provides the goals specified in the *Deployment Request*.

Note that components do not appear in this model. Components are architectural units that are packaged into artifacts. The components definitions are mapped and developed by the architect/developer offline, while components instantiation and binding are carried out in the online section.



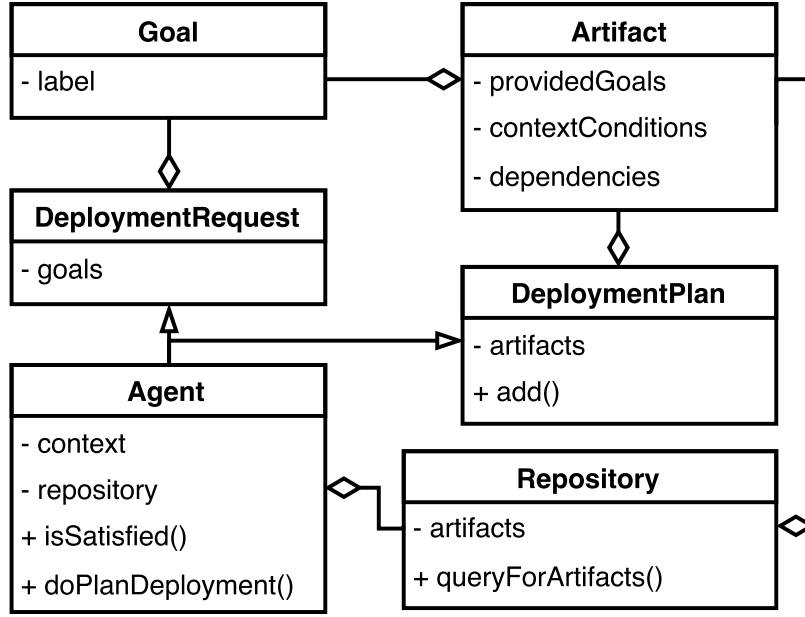


Figure 4.6: The Goalp Deployment metamodel

Artifacts are *deployable* for an agent if all its context conditions and dependencies are satisfiable. Goals are *achievable* if their artifacts are deployable as part of a deployment plan to achieve/fulfill such goals.

## Planning Method

To come up with a deployment plan for a given deployment request and context, we present Algorithm 1. It implements the *Agent*'s `doPlanDeployment` method of Goalp metamodel.

Algorithm 1 works as follows: it receives as a parameter a deployment request, which contains a list of goals. For each goal in the list, it queries the repository for artifacts that provide this goal (line 4). The repository returns a list of artifacts. For each artifact, the algorithm looks for a sub-plan for such artifact (lines 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 (lines 8-9). If the list of dependencies of the artifact is empty (line 10), then 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 these dependencies. If the results of the recursive call is not NULL (line 15), the resulting plan is added to the new plan and included into the sub-plan (lines 16-17). In both cases that new plan is added to a sub-plan, the look for a deployment plan that satisfies the selected goal is over and the inner **for** loop is halted (lines 12 and 19) and then 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**, the artifact can not be deployed. The loops continue and other artifacts will be tried. 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).

**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)

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.

### Verifying a Plan

A deployment plan is *valid* for a given context if: (i) for each artifact in the plan, all context conditions hold. (ii) there is at least one artifact in the plan that provides each dependency of each plan (i.e there is no dependency there is not satisfied).

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

Being so, we can verify if a deployment plan satisfies a deployment request by executing the following steps that verify 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 is at least one artifact that declares each intended goal and one that provides such goal.

## 4.2.2 Deployment Execution

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

To avoid static dependency between component implementations, the Dependency Injection[26] design pattern can be used. Dependency Injection is a pattern that allows for wiring together software components that were developed without the knowledge about each other. [26] The basic idea of the Dependency Injection is to have a separate object, an *assembler*, that wires together the components at runtime[26]. The client class refers to the service using its interface (the service interface). The assembler can use alternative ways to the *new* to instantiate an object so that the wiring between client objects and implementation service classes could be postponed to runtime. Using reflexive platforms we can eliminate the static dependency as the platform can find available interfaces implementations at runtime.

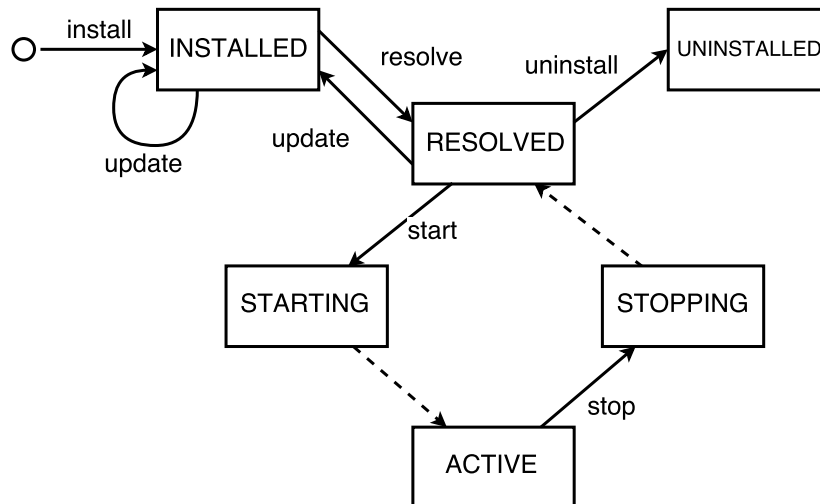


Figure 4.7: Representation of OSGi bundles lifecycle

In order to fetch and bind the components, we can make use of OSGi platform[58]. OSGi is a Java centric platform that allows dynamic fetching, binding and unbinding of

components, usually named bundles. Figure 4.7 illustrates the lifecycle of bundles in the OSGi platform[58]. Nodes represent the states of bundles and edges represent commands that can be issued to the platform. The lifecycle begins with an *install* command. This command instructs the platform to fetch the component from a repository. When the component is already in the target environment, it is *INSTALLED*. Then, the platform starts looking for the bundle dependencies. If all dependencies are *INSTALLED* the bundle is moved to the *RESOLVED* state. *RESOLVED* bundles can be started. In the starting process the component is wired to its dependencies. When the starting process is concluded, the bundle gets *ACTIVE*. The lifecycle of the bundle can come to an end by sequence of commands *stop* and *uninstall*. A bundle in states *INSTALLED* and *RESOLVED* can be updated to a newer version by the command *update*.

# Chapter 5

## Evaluation

In this chapter, we focus on the evaluation of the proposed approach. To do so we used the Goal-Question-Metric (GQM) evaluation methodology [10].

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?
  - Accurately maps 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.

In the context of heterogeneity, we can have many artifacts in the repository that provide the same goal but with different context conditions. We named variability level the number of artifacts present in the repository that provide the same goal. It can affect the scalability of the planning because it leads the algorithm to verify alternative dependency trees, which can be computing intensive.

- 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.

## 5.1 Feasibility Assessment

We validated the feasibility of the approach applying it to the Filling Station Advisor.

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. The experiments to evaluate the algorithm correctness were implemented as automated tests under Java's JUnit framework.

The code used to execute the evaluation is available on a public repository <sup>1</sup> as well as the data obtained and scripts used to treat it. <sup>2</sup>

*Question 1.1, mapping components and artifacts*

We applied the patterns described in Table 4.1 to the CGM depicted in Figure 3.1. Then we defined the artifacts that would package that components following the proposed deployment architecture style (4.1.3). We then mapped 21 different artifacts.

*Question 1.2 and 1.3*

We instantiated an artifact repository with the mapped artifacts. We defined 7 deployment scenarios under 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 are scenarios for which there is no valid deployment plan.

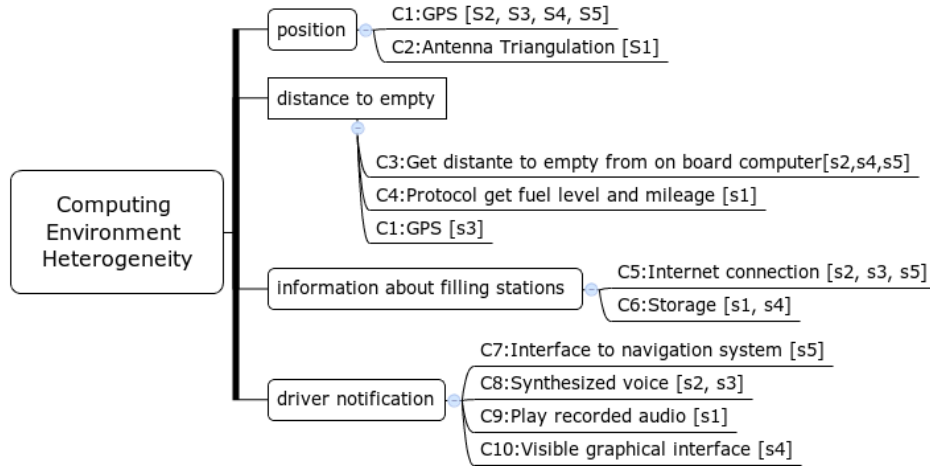


Figure 5.1: Computing Environment Evaluation Scenarios

<sup>1</sup>The evaluation experiments and step by step guide are available at: <https://github.com/lesunb/goalp> Accessed on December 4th, 2016

<sup>2</sup>The dataset obtained and the R-Script[25] used to analyze the dataset is available at: <https://github.com/lesunb/goalp-evaluation/tree/master/scalability/exp2> Accessed on December 4th, 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.

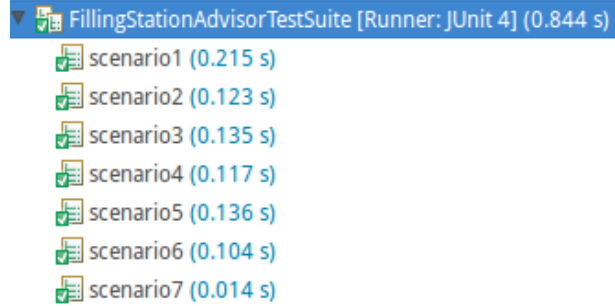


Figure 5.2: Passing Tests

*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. We executed the planning 100 time. Table 5.1 shows the scenarios, the context, time spent for planning in each scenario, in mile-seconds together with standard deviation.

Table 5.1: Time to come up with a plan

Ref.	Context	Time (ms)	Std
s1	C2, C4, C6, C9	12.28 ms	30.69
s2	C1, C3, C5, C8	6.24 ms	16.22
s3	C1, C5, C8	9.27 ms	20.62
s4	C1, C3, C6, C10	9.01 ms	20.94
s5	C1, C3, C5, C7	6.83 ms	17.18
s6	C3, C6, C8	8.74 ms	18.76
s7	C1, C3, C7	6.44 ms	17.51

## 5.2 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 a different number of artifacts were made. With this, we could evaluate the impact of the generated plan size in the planning time. The generated repository had 143,500 artifacts.

The experiments were conducted using a virtual machine in the Azure Cloud. It was used an F1 instance, with 2.4 GHz Intel Xeon® E5-2673 v3 (Haswell) processor, 2GB DDR3 1600MHz memory, and Linux (Kernel 4.4.0-47-generic). OpenJDK(1.9 64bits-build 9) was used to build and run the project.

The code used to execute the evaluation is available on a public repository <sup>3</sup> as well as the data obtained and scripts used to handle data and plot graphs. <sup>4</sup>

*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 experiment was repeated 10 times and the *observed time vs plan size* is shown in a boxplot graph in Figure 5.3.

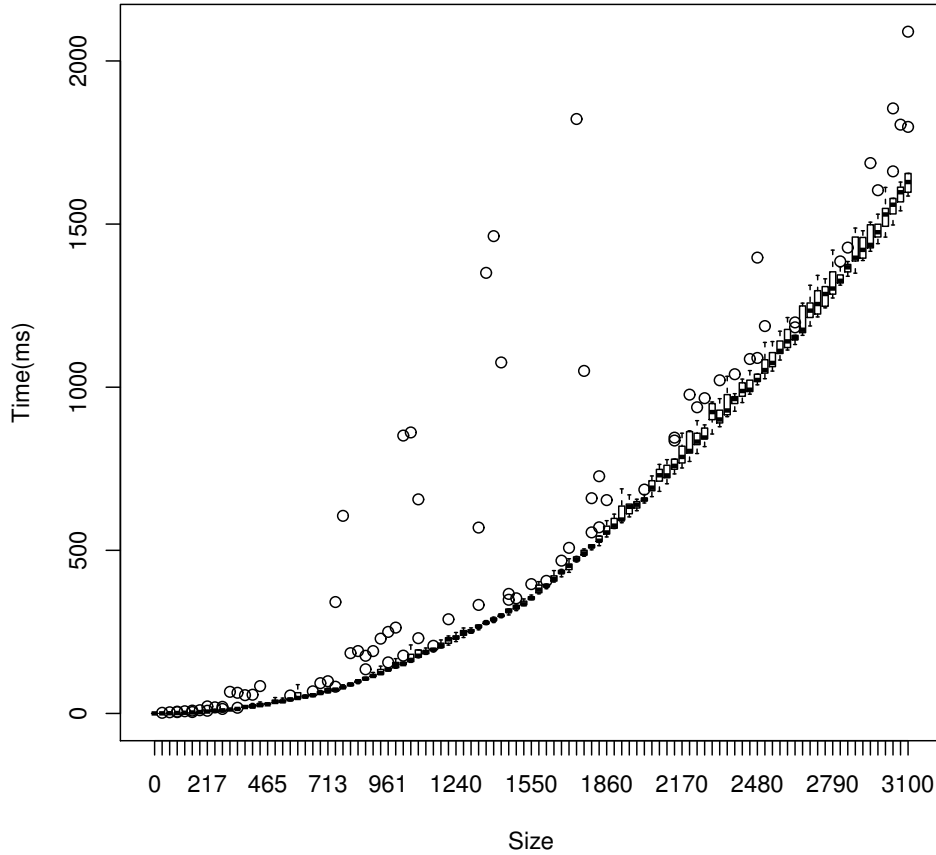


Figure 5.3: Scalability over the size of plan

*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, from 1 to 10. A variability level of 1 being so that for each plan implementation there was just one artifact that implement the plan. While in variability level 2, for each plan implementation there was two artifacts, and so on. The experiment was repeated 10 times.

<sup>3</sup>The needed source code and a step by step guide are available at <https://github.com/lesunb/goalp/tree/master/scalability-evaluation> Accessed on December 4th, 2016

<sup>4</sup>R Scripts[25] and used dataset are available at <https://github.com/lesunb/goalp/tree/master/scalability-evaluation> Accessed on December 4th, 2016



The average of the measures is depicted in Figure 5.4. Each curve represents a different level of variability.

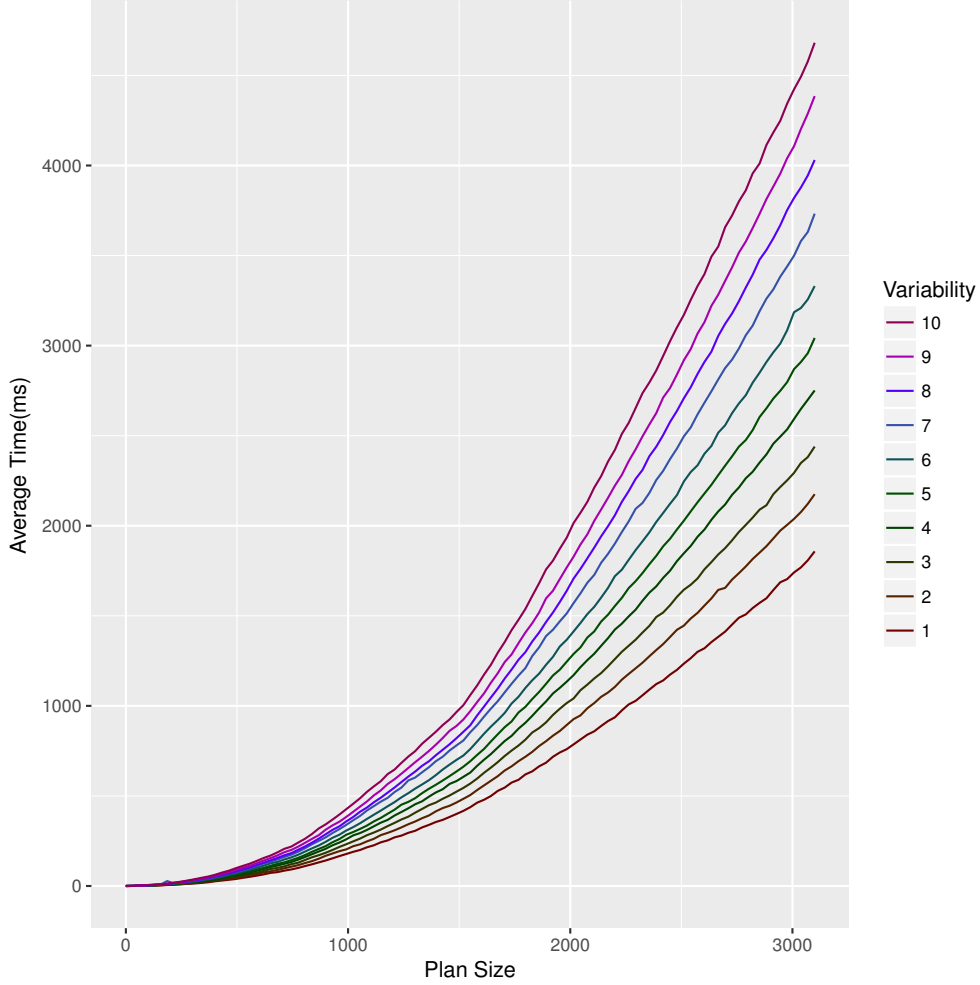


Figure 5.4: Scalability over variability level

In the worst case, a deployment request that needs 3,100 artifacts, with 10 variants for each artifact, took less than 5s to be planned. Requests that required up to 1,000 artifacts could be fulfilled in less than half-second.

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

### 5.3 Threats to validity

We recognize some threats to the validity of the evaluation:

*Construct validity:* We used GQM methodology to proper design our experiments. We did an assumption that goal can be traced to implemented component and so the artifacts. The mapping of goals and plans to their concrete counterparts in the system architecture is a well-known problem of the requirements engineering community.

*Internal validity:* The suitability of Goalp for deployment of Filling Station Advisor has been presented. The deployment planning result for the scenarios was validated. In regard to scalability, we executed each experiment in a single resource and evaluated each time a single controlled variable.

*External validity:* The scalability was evaluated for a randomly generated repository. For other repositories, the chains of dependencies could have different properties, which could change how the planning algorithm scale over the plan size. Another threat is that the scalability evaluation was conducted in a cloud environment on a reasonably powerful machine. In other scenarios, we could have a much more limited machine in relation to processor power, memory size, network bandwidth, and battery. In that case, the planning could take longer.

# Chapter 6

## Related Work

In this chapter, we highlight the most closely related work.

### 6.1 Goal Oriented Approaches

Angelopoulos et al. [5] 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 at the architecture dimension with the variability of components and implementations. However, their approach does not handle variability at deployment.

Ali et al.[2] explore the optimization of the deployment for a given context variability space in which the system will be deployed. Contextual Goal Model (CGM) was used to represent aspects of the environment related to the solution space, which was to be analyzed at design-time. This analysis at design-time can be used to evaluate which alternative strategy to implement. It differs from our work in which we explore the context of the computing environment, not the solution space. Our approach allows for, at deployment time, choosing between components already available. Both approaches could be used in tandem, as both rely on CGM but provide complementary kinds of analyses.

### 6.2 Automatic Deployment Approaches

Researchers have investigated Dynamic Software Product Line (DSPL) a way of adapt for variations in users requirements and system environments. DSPLs extend the concept of conventional Software Product Lines (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.[12]

Bencomo et al. [13] use a SPL approach 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 selected components. 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 on the deployment itself. Mizouni et al. [42] use a feature model associated with context requirements.

Leite et al. [23] propose 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 the case of failure the machine can be restarted or terminated and then a new one created. The approach is specific to cloud environments and requires instantiating at design-time a model knowledgeable 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, especially in unknown environments.

Gunalp et al. [28] presents an approach for automatic deployment, in which the deployment specialist specifies the system deployment in terms of resources and desired target states of such resources. The approach follows preset strategies to keep the managed software resources in the specified states. They use a low-level model to drive the adaptation: implemented strategies to move watched resources to target states. Differently, our approach uses a goal model which is a more abstract model.

## 6.3 Package managers

Package managers, such as Debian package manager[6], are capable of solving dependencies and deploying software, however, their approach to heterogeneity is limited. In Debian package manager, the heterogeneity is handled for processor architecture and version of operating system. There are separated repositories for each specification of architecture and operating system. The users' machines have only repositories registered with compatible repositories. In our approach, separated repositories are not required, context conditions are evaluated for each artifact instead. Another difference is in how dependencies are declared. 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, for advisor root strategy we create an artifact that declares *Get Position* as a requirement, but not a specific artifact is required. Any artifact that provides *Get Position* would satisfy the requirement. This requirement declaration, in terms of interface, associated with context conditions allow for a more flexible dependency resolution at deployment-time.

Table 6.1 summarizes work most related to Goalp. Ali et al.[2] and Angelopoulos et al. [5] works are both goal-oriented works that handle variability at the design of a system, however, they do not handle heterogeneity in the computing environment and has no support adaptation in the deployment time. Mizouni et al. [42], Leite et al. [23] and Gunalp et al.[28] works' handle some kind of adaptation at deployment time but uses a low-level model to drive the adaptation which requires a knowledge about the computing environment at design-time.

Table 6.1: Comparing characteristic properties of selected approaches related to Goalp

Work by	Goal Oriented	Handle Heterogeneity	Autonomic Deployment
Ali et al.[2]	Yes	No	No
Angelopoulos et al. [5]	Yes	No	No
Mizouni et al. [42]	No	Yes	No
Leite et al. [23]	No	Yes	No
Gunalp et al.[28]	No	Yes	No
Goalp	Yes	Yes	Yes

# Chapter 7

## Conclusion

### 7.1 Conclusion and future work

In this work, 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 solving 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 at 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 from 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 show that the algorithm is capable of coming up with a plan, in a reasonably large scenario in few 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 adapt 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[33] architecture. We should note that this work aims at identifying a single valid deployment plan, as long as it exists. However, it is out of scope of our current work to find the best valid plan in case multiple valid plans exist. In future work, a CSP approach

might integrate our deployment plan algorithm to address such issue. 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 an 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, 16
- [2] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Requirements-driven Deployment. *Softw. Syst. Model.*, 13(1):433–456, February 2014. 36, 37, 38
- [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, 17
- [4] Andersson, Jesper. A deployment system for pervasive computing. In *Proceedings 2000 International Conference on Software Maintenance*, pages 262–270, 2000. 1
- [5] 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, 18, 36, 37, 38
- [6] Osamu Aoki. Debian Manual Chapter 2. Debian package management, 2016. 10, 37
- [7] Apache. Apache Maven Project, 2016. 10
- [8] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Comput. Netw.*, 54(15):2787–2805, October 2010. 1
- [9] 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
- [10] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley, 1994. 30
- [11] 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



- [12] 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. 36
- [13] 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, 36
- [14] 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
- [15] 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. 8
- [16] 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
- [17] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimburger, André van der Hoek, and Alexander L. Wolf. A Characterization Framework for Software Deployment Technologies. Technical report, 1998. 1, 9
- [18] Chocolatey. Chocolatey - The package manager for Windows, 2016. 10
- [19] Ilica Crnkovic, Judith Stafford, and Clemens Szyperski. Software Components beyond Programming: From Routines to Services. *IEEE Software*, 28(3):22–26, 2011. 11
- [20] 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. 8, 9, 19
- [21] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, April 1993. 3, 12
- [22] 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
- [23] Alessandro Ferreira Leite. *A user centered and autonomic multi-cloud architecture for high performance computing applications*. PhD thesis, Paris 11, 2014. 10, 37, 38
- [24] Anthony Finkelstein and Andrea Savigni. A framework for requirements engineering for context-aware services. 2001. iv, 5
- [25] The R Foundation. The R Project for Statistical Computing, 2016. 31, 33

- [26] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. 13, 28
- [27] 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. 11
- [28] Ozan Gunalp, Clement Escoffier, and Philippe Lalanda. Rondo A Tool Suite for Continuous Deployment in Dynamic Environments. pages 720–727. IEEE, June 2015. 10, 37, 38
- [29] George T. Heineman and William T. Council, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 10, 11
- [30] Homebrew. Homebrew, The missing package manager for macOS, 2016. 10
- [31] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010. 9, 10
- [32] 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
- [33] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. 6, 7, 39
- [34] 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
- [35] 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
- [36] 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
- [37] Robbert Laddaga. Self Adaptive Software SOL BAA 98 12. Technical report, 1997. 6
- [38] Pattie Maes. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM, 1987. 6
- [39] 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

- [40] Danilo Mendonça. Dependability Verification for Contextual/Runtime Goal Modelling, 2015. 9
- [41] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, September 2012. 1
- [42] 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. 37, 38
- [43] 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
- [44] 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
- [45] 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
- [46] 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
- [47] 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
- [48] 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
- [49] 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. 11
- [50] 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

- [51] 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
- [52] 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
- [53] D. Spinellis. Don’t Install Software by Hand. *Software, IEEE*, 29(4):86–87, July 2012. 1
- [54] D. Spinellis. Package Management Systems. *Software, IEEE*, 29(2):84–86, March 2012. 10
- [55] Maxim Svistunov, Stephen Wadeley, and Tomáš Čapek. Red Hat Enterprise Linux 6 - Chapter 8. Yum, 2016. 10
- [56] 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
- [57] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. 11
- [58] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.1*. 2007. 12, 28, 29
- [59] Axel Van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures*, pages 25–43. Springer, 2003. 2, 12
- [60] 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
- [61] 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, 18, 19