Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Autonomic Goal-Driven Deployment in Heterogeneous Computing Environments

Gabriel Siqueira Rodrigues

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientadora
Prof.ª Dr.ª Genaina Nunes Rodrigues

Brasília
2016

CIP — Catalogação Internacional na Publicação

Rodrigues, Gabriel Siqueira.

Autonomic Goal-Driven Deployment in Heterogeneous Computing Environments / Gabriel Siqueira Rodrigues. Brasília : UnB, 2016.
103 p. : il. ; 29,5 cm.

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

1. Engenharia de requisitos, 2. Variabilidade Arquitetural,
3. Ambientes Heterogêneos, 4. Implantação Automatizada

CDU 004.4

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

Brasília, 16 de Dezembro de 2016

Agradecimentos

Firstly, I would like to express my gratitude to my advisor, Genaína, for her enthusiasm, patience and wisdom. Which was fundamental to the conclusion of this work. And to the members of our research group Danilo Mendonça and Felipe Pontes.

Resumo

Vemos um crescente interesse em aplicações que devem contar com ambientes de computação heterogêneos, como a Internet das Coisas (IoT). Esses aplicativos são destinados a executar em uma ampla gama de dispositivos com diferentes recursos computacionais disponíveis. Para lidar com algum tipo de heterogeneidade, como dois tipos possíveis de processadores gráficos em um computador pessoal, podemos usar abordagens simples como um script que escolhe a biblioteca de software certa a ser copiada para uma pasta.

Essas abordagens simples são centralizadas e criadas em tempo de design. Eles requerem um especialista ou equipe para controlar todo o espaço de variabilidade. Dessa forma, essas abordagens não são escaláveis para ambientes altamente heterogêneos. Em ambientes altamente heterogêneos, é difícil prever o ambiente computacional em tempo de projeto, implicando provavelmente indecidibilidade na configuração correta para cada ambiente. Em nosso trabalho, propomos GoalD: um método que permite a implantação autônoma de sistemas, refletindo sobre os objetivos do sistema e seu ambiente computacional. Por implantação autônoma, queremos dizer que o sistema é capaz de encontrar o conjunto correto de componentes para o ambiente computacional alvo, sem intervenção humana.

Nós avaliamos nossa abordagem em um estudo de caso: conselheiro de estação de abastecimento, onde uma aplicação aconselha um motorista onde reabastecer / recarregar seu veículo. Nós projetamos a aplicação com variabilidade em nível de requisitos, arquitetura e implantação, o que pode permitir que a aplicação projetada seja executada em diferentes dispositivos. Para cenários com diferentes ambientes, foi possível planejar a implantação de forma autônoma. Além disso, a escalabilidade do algoritmo que planeja a implantação foi avaliada em um ambiente simulado. Os resultados mostram que usando a abordagem é possível planejar de forma autônoma a implantação de um sistema com milhares de componentes em poucos segundos.

Palavras-chave: Engenharia de requisitos, Variabilidade Arquitetural, Ambientes Heterogêneos, Implantação Automatizada

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 GoalD: 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: Requirements Engineering, Architecture Variability, Heterogeneous Environments, Automated Deployment

Contents

1 Introduction **1**

1.1 Problem Definition 1

1.2 Proposed Solution 3

1.3 Contributions Summary 3

1.4 Structure 4

2 Background **5**

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

3 Filling Station Advisor **14**

3.1 Motivating Example: The Filling Station Advisor 14

4 The GoalD Approach **17**

4.1 Offline 18

4.1.1 Goal Modelling 18

4.1.2 From Goals to Components Specification 19

4.1.3 Packaging Components into Artifacts 21

4.1.4 Development Process 24

4.1.5 Activities 24

4.2 Online 25

4.2.1 Automated Deployment Planning 26

4.2.2 Deployment Execution 28

5 Evaluation **31**

5.1 Feasibility Assessment 32

5.2 Scalability Assessment 34

5.3 Threats to validity 35

6 Related Work **37**

7 Conclusion **39**
7.1 Conclusion and future work 39

Referências **41**

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 of the steps in GoalD 17
4.2 Artifact Structure 22
4.3 Dependency graph 23
4.4 Roles collaboration 24
4.5 Deployment Process Activities 25
4.6 Goald Autonomic Deployment 26
4.7 The GoalD Deployment metamodel 27
4.8 Representation of OSGi bundles lifecycle 30

5.1 Computing Environment Evaluation Scenarios 33
5.2 Passing Tests 33
5.3 Scalability over the size of plan 34
5.4 Scalability over variability level 35

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 an environment a highly heterogeneous computing environment as it is formed by different sets of devices, with different resources, and which are only partially known at design-time. Ubiquitous Computing [11], Internet of Things (IoT) [8], Assisted Living [34] and Opportunistic Computing [53] are examples of computing architectures that have to be typically designed for a highly heterogeneous computing environment.

Software deployment is the process of getting a software ready to be used in a given computing environment[16]. 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[42]. 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[54]. 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 architec/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 [35][46][47][48][60]. Some works in the literature have investigated variability in goal models with adaptation purpose [5][63]. 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 GoalD: 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[15][21][62]. 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.

GoalD 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. GoalD 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 GoalD. 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.

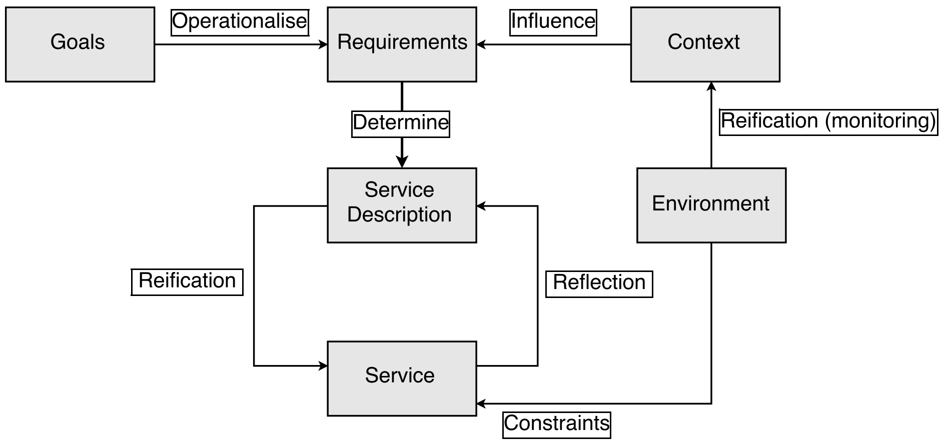


Figure 2.1: Context-aware services framework by [24]

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 [13], a system should keep software requirements model at runtime, and use such model to drive the system adaptation.

2.2 Self-Adaptive Systems

Self-adaptivesses is an approach in which the system *"evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible."*[36]. 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[51].

A key concept in self-adaptive systems is the awareness of the system. It has two aspects[51]:

- *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.[33] 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".

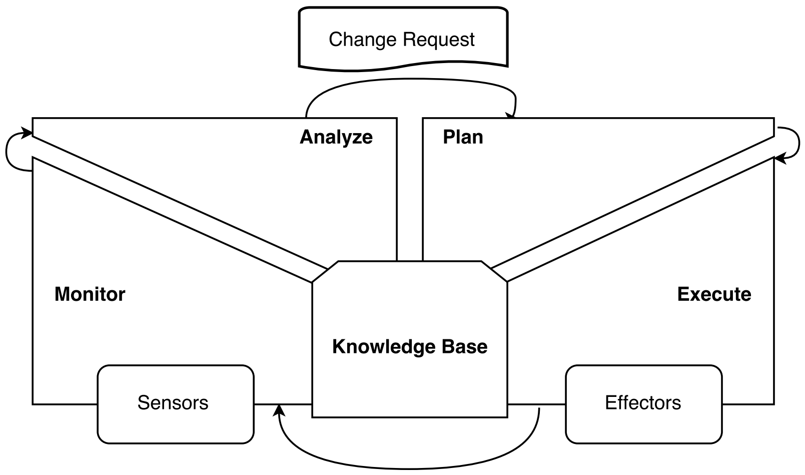


Figure 2.2: MAPE-K Reference Architecture

MAPE-K is a reference architecture originally proposed for autonomic computing [32] 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, them 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.

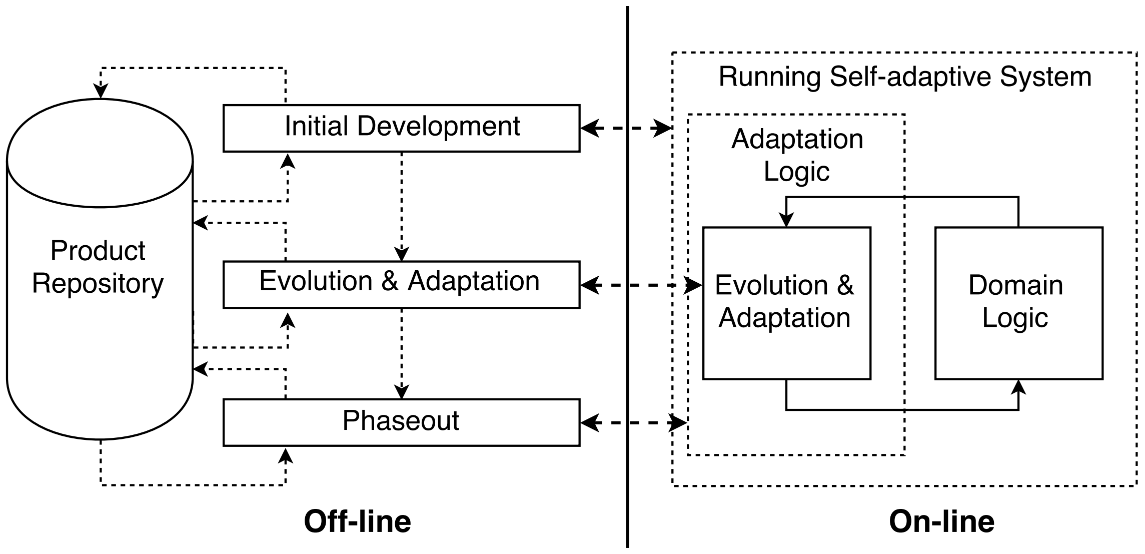


Figure 2.3: A Life-cycle model for Self-Adaptive Software System[3]

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 [32].

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 [45]. Goals capture the various objectives the system under consideration should achieve. In particular, Tropos[15] 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* [62] which proposes the concepts of actor, goal, plan, resource and social dependency to model both the system-to-be and its organizational operating environment [15] [44].

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 [14]. 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 [63], behavior [20], probability of achieving success [41] and achievability of goals [49].

Salehie et al. [52] 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. [41] 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[16]. 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*[30] environment applies automation in building and getting components ready

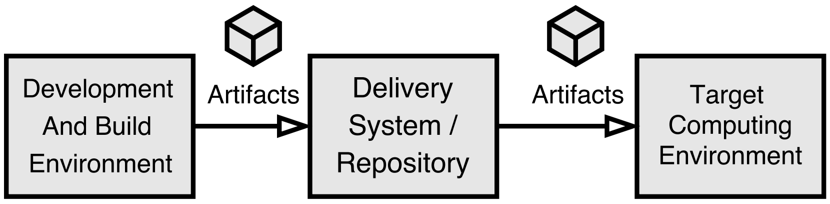

Development And Build Environment
Artifacts
Delivery System / Repository
Artifacts
Target Computing Environment

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 [55]. 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 [27], and changes in the context of operation [12].

Continuous delivery[30] 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 aptitute/apt-get(Debian based Linux distributions) [6], yum (Red Hat based Linux distributions) [56], Homebrew (MacOS)[29] and Chocolatey (Windows)[17] 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"[28].

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.[58].

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.[28] 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[31]. The architectural view of a software can be formalized via an architecture description language (ADL)[40].

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[26] 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[50] 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 [57] 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 [57]. Flashmod uses a three-layer model: goals, management and components proposed by Kramer and Magee [35], extending it to allow distributed agreement in a given configuration.

OSGi[59] 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 [60] 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. [48] 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. [63] 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. [25]

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 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[25]. 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.

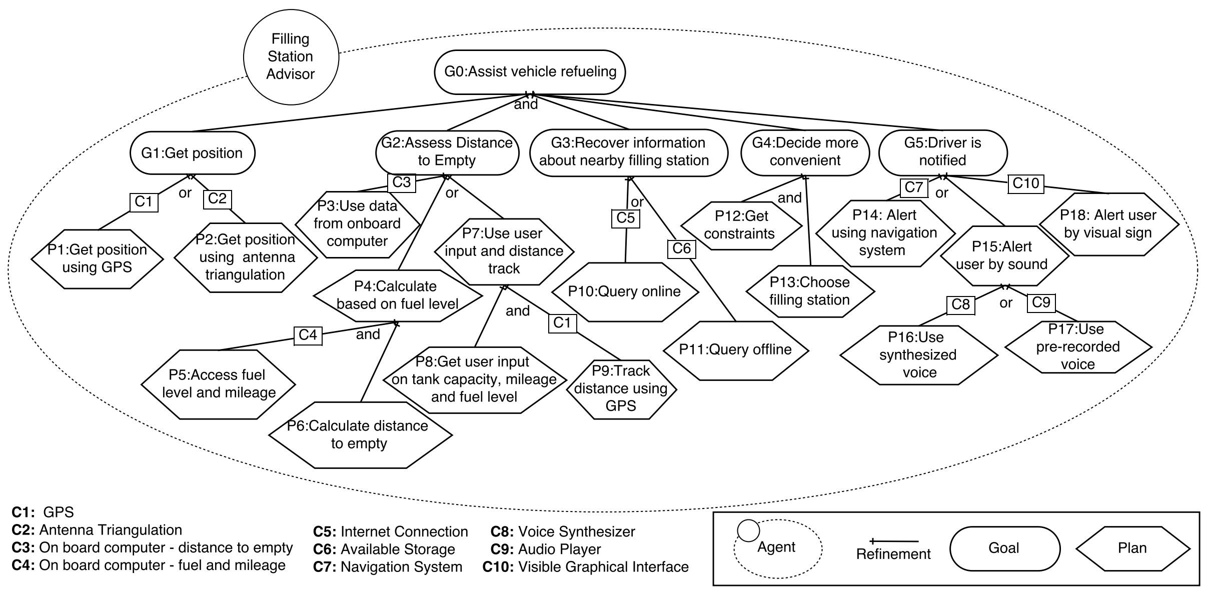

Filling Station Advisor
G0:Assist vehicle refueling
and
G1:Get position
G2:Assess Distance to Empty
G3:Recover information about nearby filling station
G4:Decide more convenient
G5:Driver is notified
C1
or
C2
P1:Get position using GPS
P2:Get position using antenna triangulation
C3
or
P3:Use data from onboard computer
P7:Use user input and distance track
P4:Calculate based on fuel level
C4
and
P5:Access fuel level and mileage
P6:Calculate distance to empty
and
C1
P8:Get user input on tank capacity, mileage and fuel level
P9:Track distance using GPS
or
C5
C6
P10:Query online
P11:Query offline
and
P12:Get constraints
P13:Choose filling station
C7
or
C10
P14: Alert using navigation system
P15:Alert user by sound
P18: Alert user by visual sign
C8
or
C9
P16:Use synthesized voice
P17:Use pre-recorded voice
C1: GPS
C2: Antenna Triangulation
C3: On board computer - distance to empty
C4: On board computer - fuel and mileage
C5: Internet Connection
C6: Available Storage
C7: Navigation System
C8: Voice Synthesizer
C9: Audio Player
C10: Visible Graphical Interface
Agent
Refinement
Goal
Plan

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: Assist Vehicle Refueling* 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.

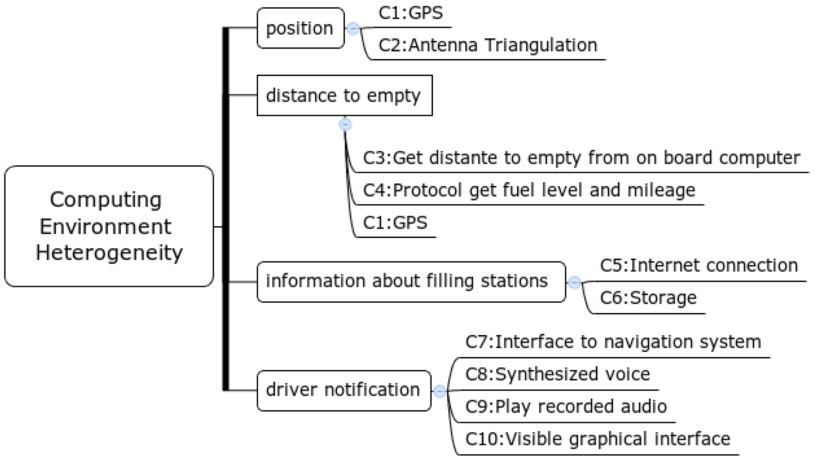

Computing Environment Heterogeneity
position
C1:GPS
C2:Antenna Triangulation
distance to empty
C3:Get distante to empty from on board computer
C4:Protocol get fuel level and mileage
C1:GPS
information about filling stations
C5:Internet connection
C6:Storage
driver notification
C7:Interface to navigation system
C8:Synthesized voice
C9:Play recorded audio
C10:Visible graphical interface

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 GoalD Approach

Our approach to goal-driven automated deployment (GoalD) is divided into *offline* and *online* activities. This differentiation is in line with established concepts on software development processes for variability configuration and adaptive systems (e.g., [3]). *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 GoalD.

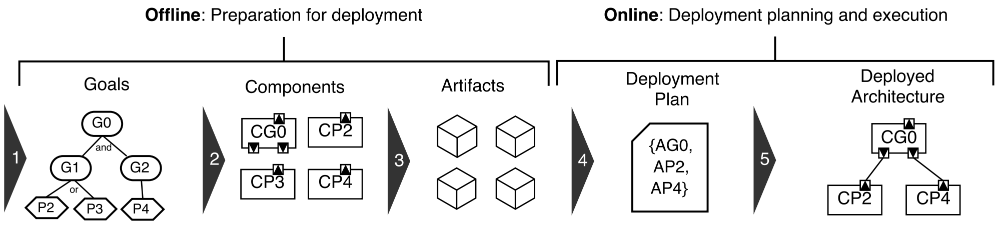


Figure 4.1: Overview of the steps in GoalD

The *offline* activities are conducted by software engineers and include the capturing of requirements and the development and publishing of software components to achieve them. In GoalD, the offline activities consist of: (i) requirements modeling as goals (Section 4.1.1), (ii) goal mapping to components (Section 4.1.2), and (iii) components packaging into artifacts, which are then published to a components repository (Section 4.1.3).

The *online* activities concern the selection and application of the components that fit the host environment. In GoalD, we aim to make this activity supported by automated reasoning and potentially embedded in the software itself. The online activities in GoalD include: (i) deployment planning, which includes automated reasoning and decision on the artifacts suitable for the target computing environment so that goals depicted in the offline activities are achieved (Section 4.2.1), and (ii) architecture deployment based on artifacts fetching and binding (Section 4.2.2).

In the next sections we further explain each of the steps in detail.

4.1 Offline

Previously, goal-driven approaches were proposed for introducing variability at requirements, context modeling, software behavior, and software architecture[5][63]. 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

The modelling structure of GoalD is the Deployment Goal Model (DGM) a specialized version of CGM with *resource* notion, that specifies restrictions to the applicability of plans in relation to the deployment environment.

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.*

In GoalD, the context as a description of the target computing environment is concretized 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 (c1), Internet connection (c5), and voice synthesizing (c8) are capabilities that should be available in the computing environment to have their related plans executed. For example, the execution of plan *P1:Get position using GPS* requires GPS (c1) to be available.

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.

Henceforth, we will refer to DGM as our underlying goal model structure. We formally define the DGM as follows:

Definition 2 (Deployment Goal Model) *A Deployment Goal Model (DGM) is a tuple (M, env_res, ctx_cond) where:*

- *M is a design-time goal model 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.*
- *env_res is a set of environment resources.*
- *ctx_cond: R → env_res associates a relationships in R with a resource or more in env_res.*

Context conditions are restrictions to the applicability of a plan and are used to solve variability at deployment. A context condition (ctx_cond) is satisfied if its associated environment resource is present in the context. For example, in a scenario with context ctx=[c1, c5], the context condition c1 is satisfied. 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.

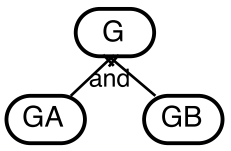
4.1.2 From Goals to Components Specification

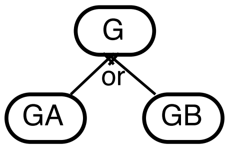
Components are units of composition with contractually specified interfaces and explicit context dependencies [18]. Components and interfaces can be described using architecture description languages (ADLs). Yu et al. [63] adapt the Darwin ADL [39] with elements borrowed from one of its extensions, namely Koala [61]. They also provide a method for relating goals with components, but without taking into account context conditions. In GoalD, we define patterns to map CGM elements into architectural elements, which are specified as ADL elements mostly borrowed from Yu et al. proposal [63].

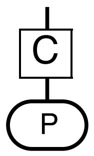
The patterns present in Table 4.1 are used to map components based on the DGM of the system. By mapping components we mean identifying which component should be developed in order to reflect the DGM of the system. By using the proposed patterns, the variability present in the DGM is kept at the architecture of the system. Theses patterns are an extension of Yu et al.[63] patterns for the Goals-Component view. We extended Yu et al.[63] patterns with context conditions. The presented patterns are described using goals but they can be applied for goals and plans without distinction.

An AND-refinement results in a components specification that realizes the achievement of the parent goal by achieving all its subgoals. For an AND-refinement, we define an AND-pattern applied as follows: (i) an interface specification for each node and (ii) a component specification that *provides* the node interface and *requires* each interface generated for each subnode of the AND-refinement.

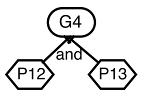
Table 4.1: Contextual Goal Model to components - patterns

And-Refinement	 `Component CG {`
	`  provides IG;`
	`  requires`
	`     IGA, IGB;`
	`}`

Or-Refinement	 `Component CGA {`
	`  provides IG;`
	`}`
	`Component CGB {`
	`  provides IG;`
	`}`

Context-condition	 `Component CG {`
	`  provides IG;`
	`  condition C;`
	`}`

The latter component implements a strategy to achieve its provided goal. It coordinates the more specific subgoals by calling them and passing one result as the input to another, when applicable. As an example, applying AND-refinement patterns for goal *G4* of the Filling Station Advisor application, will result in the specifications of the interface and its respective component `DecideMoreConvenient` and `DecideMoreConvenientImpl`, requiring both interfaces `GetConstraints` (from plan *P12*) and `ChooseFillingStation` (from plan *P13*). The resulting specification follows:

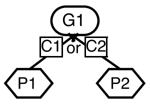


```
interface DecideMoreConvenient {}
interface GetConstraints {}
interface ChooseFillingStation {}

component DecideMoreConvenientImpl
{
  provides DecideMoreConvenient;
  requires GetConstraints,
           ChooseFillingStation;
}
```

An OR-refinement should result in one interface specification and multiple components specifications that provide the implementation of such an interface. In GoalD, the OR-pattern is applied as follows: (i) an interface specification for the OR-refined node and (ii) a component specification for each subnode *providing* the interface of the OR-refined node. For example, in the Filling Station Advisor, applying the patterns for *G1*, *P1* and

P2 will result in the specification of the interface GetPosition and of the two components GetPositionUsingGPS and GetPositionUsingAntenna, both providing the interface GetPosition. The resulting specification is as follows:



```
interface GetPosition {}
component GetPositionUsingGPS
{
  provides GetPosition;
  condition c1;
}
component GetPositionUsingAntenna
{
  provides GetPosition;
  condition c2;
}
```

Differently from AND-refinements, components derived from subgoals of OR-refinements provide, i.e. implement, the interface derived from their parent node. It means that each of such implementations represents an architecture variability. At the architecture level, such variability reflects as a decoupling of their respective components, following the proposed OR-pattern.

In CGM, AND/OR-refinements can be associated with context conditions present in the CGM refinements. Such conditions are propagated to the body of the mapped components as a condition element. This is the case in the OR-refinement of *G1* via *P1* and *P2* where the plans are associated with the context conditions $c1$ and $c2$, respectively. In particular, context conditions associated to OR-refinements restrict the applicability of the alternative strategies to the availability of different resources in the computing environment.

Using the association of an AND/OR-refinement and context conditions in the component analysis, we extend the variability present in the CGM and its conditions to the architecture of the system. For example, component GetPositionUsingGPS and GetPositionUsingAntenna provide the same goal but have different context conditions ($c1$ and $c2$). It means that we can achieve the same goal by deploying one of the two component variants given that the resources are available.

By using the proposed patterns, the variability presented in the goal model is catered for in the architecture of the system. Such variability in the architecture enables adaptation to the heterogeneity in the target environment. Specified components and interfaces should then be implemented as concrete components. For example, in the Java platform, interfaces and components specified in an ADL could be implemented as Java interfaces and classes.

4.1.3 Packaging Components into Artifacts

In the architecture level, goals in the CGM are contractually specified as interfaces, which in turn, are implemented by components. From the deployment point of view, components and interfaces represent deployment units and should be packaged as files for distribution. We call them *artifacts*. To allow automated deployment, GoalD artifacts include packaged components and interfaces as well as metadata that describe the packaged elements, which can be either components or interfaces, as depicted in Figure 4.2.

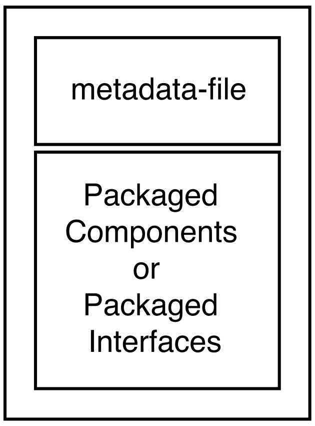


Figure 4.2: Artifact Structure

Artifacts metadata describe the artifact's goals, dependencies, and context conditions and are characterized by the following information: (i) `name`, (ii) `conditions`, (iii) `defines`, (iv) `implements` and (v) `depends`. Firstly, `name` metadata uniquely identifies an artifact. Secondly, `conditions` metadata describes the context conditions needed to deploy an artifact to a given environment. Lastly, `defines`, `implements`, and `depends` metadata are specified in terms of goals and used to analyze contractual responsibilities between artifacts: `defines` declare that an artifact *defines* the contract for a set of goals; `implements` declares that an artifact provides *implementation* for a set of goals (i.e. implements the contract declared for the goal); `depends` declares that an artifact has a dependency relationship towards artifacts that *define* and *implement* each of a set of goals.

All components packaged together in an artifact will be delivered together. In order to favor low coupling in the GoalD approach, components and interfaces should be packaged in separated artifacts, so they can be delivered only to environments where they are required and further used. In order to maximize the flexibility of systems following the GoalD deployment approach, we package interfaces and components in two respective artifact types: *definition* and *implementation*.

A *definition* artifact packages interfaces derived from the application of the patterns presented in Section 4.1.2 and, via its metadata, it declares the set of goals it provides a definition for. In a *definition* artifact, the metadata `defines` contains a list of goals it provides definition for as packaged interfaces, i.e. contracts. For example, the interface definition for goal *G1* of the Filling Station Advisor, namely `GetPosition`, is packaged in a *definition* artifact along with the following metadata:

```
name: GetPosition.def
defines: GetPosition
```

The *implementation* artifact type packages components, where the `implements` metadata contains the list of goals provided by the packaged components of the artifact, specified in accordance with those AND/OR-patterns in Section 4.1.2. The `depends` metadata contains a list of goals that packaged components depends on. For example, the component for goal *G4* of the Filling Station Advisor, namely `DecideMoreConvenient`, is packaged in an *implementation* artifact along with the following metadata:

```
name: DecideMoreConvenient.impl
implements: DecideMoreConvenient
depends:GetConstraints,ChooseFillingStation;
```

Note that `DecideMoreConvenient.impl` depends on the definition and implementation of goals `GetConstraints` and `ChooseFillingStation`. Therefore, at deployment time, components that `defines` and `implements` goals `GetConstraints` and `ChooseFillingStation` should be included in order to successfully deploy `DecideMoreConvenient.impl` artifact.

Finally, the `condition` metadata of *implementation* artifacts reflects the context conditions of packaged components. For example, in the Filling Station Advisor, the components `GetPositionUsingGPS` and `GetPositionUsingAntenna` are packaged into separate artifacts with the following metadata:

```
name: GetPositionUsingGPS.impl
implements: GetPosition
conditions: c1

name: GetPositionUsingAntenna.impl
implements: GetPosition
conditions: c2
```

`GetPositionUsingGPS` and `GetPositionUsingAntenna` are artifacts that implement the same goal `GetPosition` and can be deployed in different contexts, keeping at deployment level the variability introduced by the CGM as well as the decoupling following the patterns applied in previous Section 4.1.2.

Artifacts forms dependency trees. Figure 4.3 depicts the dependency relationship between artifacts. A1 is a root interface and have 3 dependencies, that are provides by A2, A3 and A4. A2 and A3 have one common dependency, provided by A5. A5 and A6 have no dependencies.

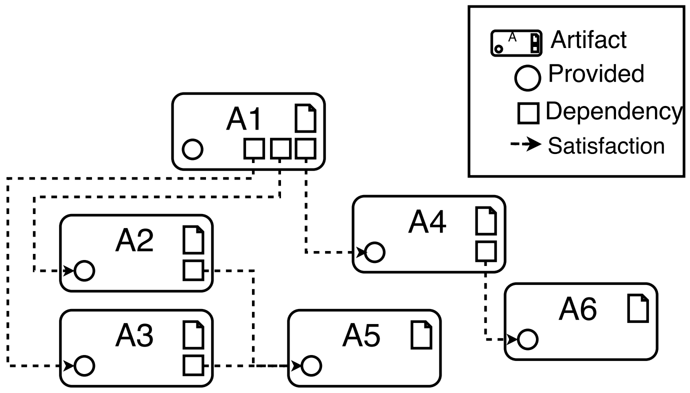


Figure 4.3: Dependency graph

After components and their corresponding metadata information are packaged accordingly into artifacts, they are registered in a repository so that they can be distributed to the target environment. In the registration process, an artifact is uploaded to the repository, its metadata is processed and registered in the repository database, where they can be queried in the next step of GoalD: the deployment planning.

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 4.4 summarizes the collaboration between the roles.

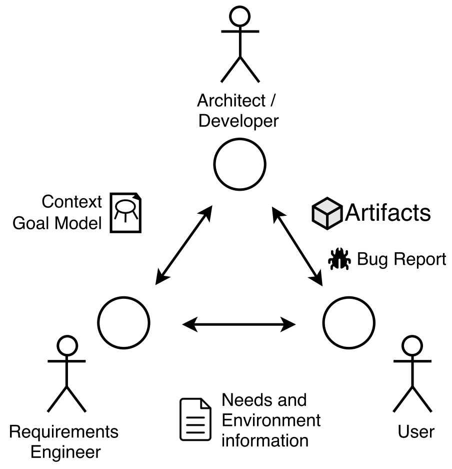


Figure 4.4: Roles collaboration

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 4.5 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

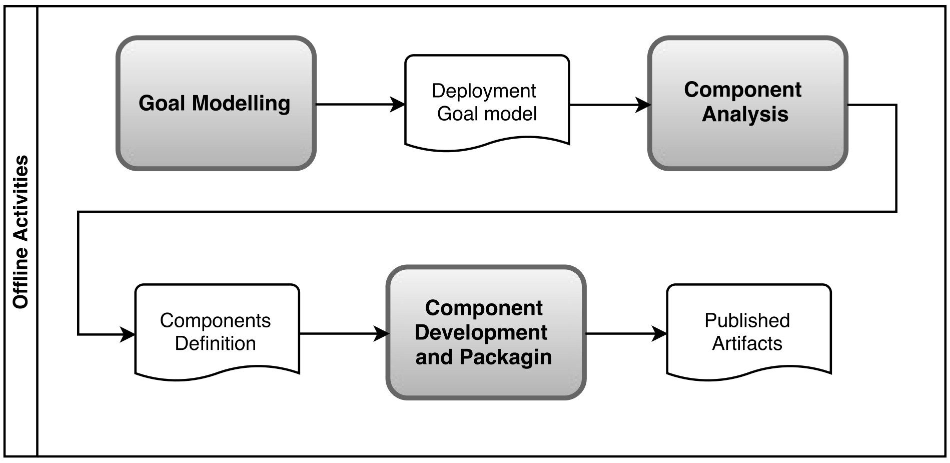


Figure 4.5: Deployment Process Activities

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 4.6 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 deploymeng 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.

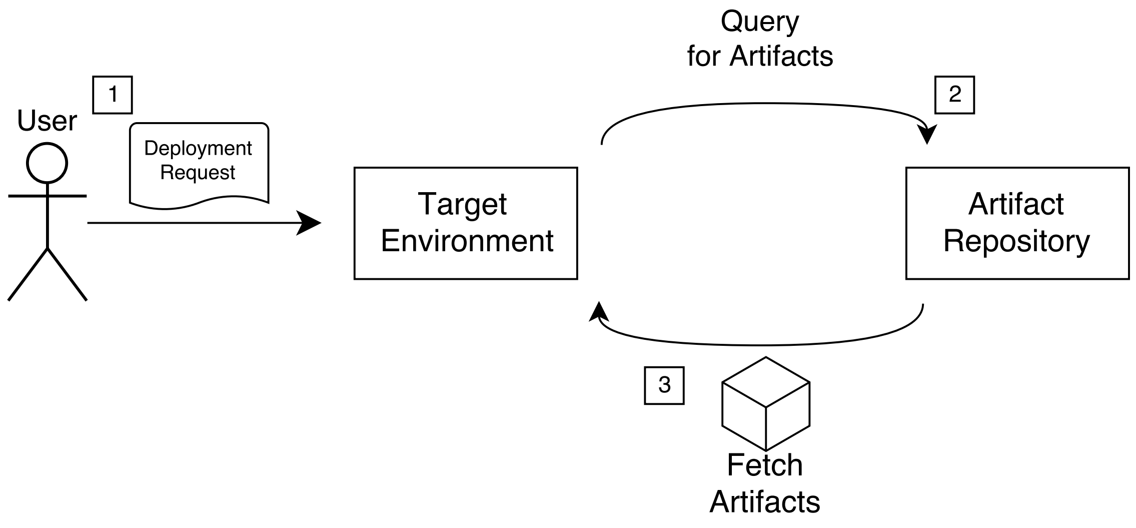


Figure 4.6: Goald Autonomic Deployment

After components and their corresponding metadata information are packaged accordingly into artifacts, they are registered in a repository so that they can be distributed to the target environment. In the registration process, an artifact is uploaded to the repository, its metadata is processed and registered in the repository database, where they can be queried in the next step of GoalD: the deployment planning.

4.2.1 Automated Deployment Planning

In the online part of the approach, the focus is on the automated deployment planning of the artifacts present in the components repository. In order to carry out such planning, the stakeholders explicitly define a set of goals to be achieved in a computing environment as a deployment request. Then, the target environment realizes a deployment planning by analyzing the available computing resources and artifacts present in the repository. The deployment plan consists of a selection of artifacts that can allow for the goals achievement relying on the available computing environment. Finally, the deployment is executed by fetching the selected artifacts from the repositories and binding them.

Prior to presenting GoalD's approach to automated deployment planning, we present GoalD's deployment metamodel in the following Section 4.2.1 as the underlying structure that defines major conceptual elements of GoalD's automated deployment. Then we present the algorithm in Section 4.2.1 to automate the deployment planning.

Metamodel

The metamodel of GoalD consists of six major elements: (1) *Goal*, (2)*Artifact*, (3)*Agent*, (4)*Repository*, (5)*Deployment Request*, and (6)*Deployment Plan*. Figure 4.7 presents the GoalD metamodel.

Artifact is the central entity at deployment level. As described in Section 4.1.3, artifacts have *conditions*, *defines*, *implements*, and *depends* which create inter-relations amongst artifacts, so that an artifact that has a goal dependency is dependent on an artifact that provides *definition* and *implementation* for such a goal. An artifact is *deployable* 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 such goals. The *Agent* can accept deployment requests, an action that should trigger the deployment planning. An agent knows the *repository* where it looks for artifacts. A *repository* has a set of artifacts that it can be queried about by the `queryForArtifacts` method, which receives a goal as argument and returns all artifacts in the repository providing that goal. An *agent* can verify *conditions* of an artifact by evoking the *isSatisfied* method for it. 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*. Finally, a *Deployment Plan* is composed of a set of artifacts that realizes a set of goals in a

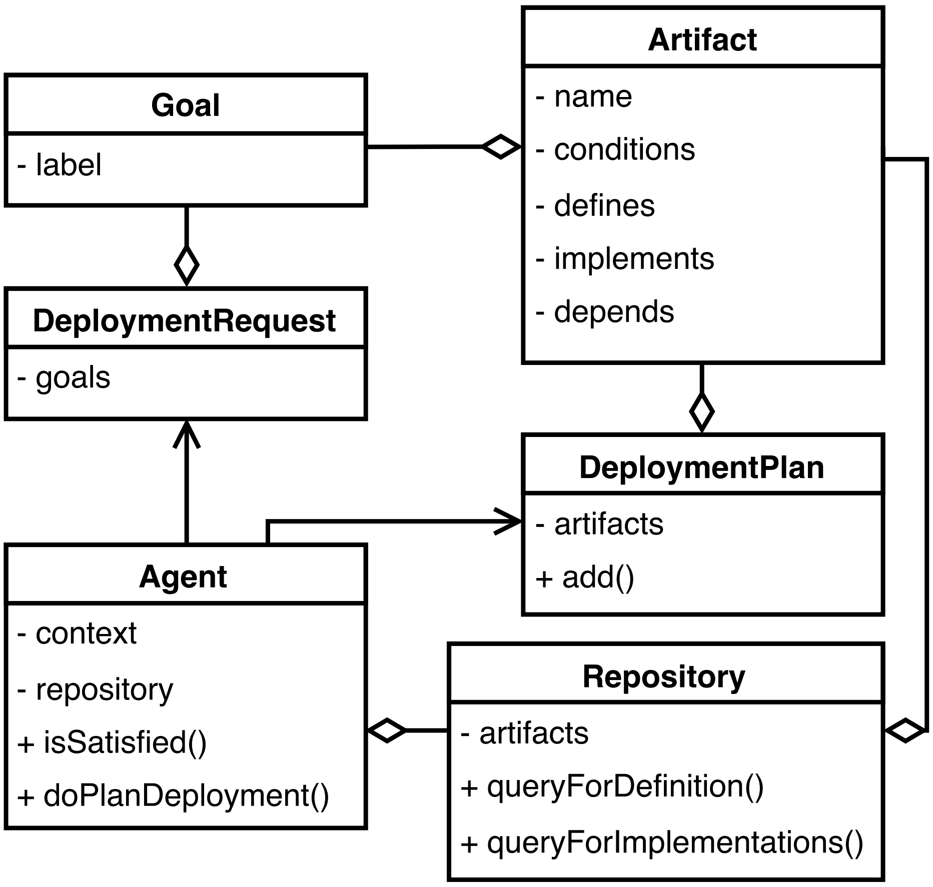


Figure 4.7: The GoalD Deployment metamodel

certain computing environment. The plan of such deployment is devised by the algorithm further presented in the next section.

Planning Algorithm

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 the GoalD metamodel.

Algorithm 1 works as follows: it receives a deployment request as a parameter, which contains a list of goals. For each goal in the list, it queries the repository for an artifact that provides definition of the goal (line 4). If no definition is found, the algorithm returns NULL (line 6). Otherwise, the returned definition is included in a sub-plan (line 9) and the repository in queried for implementations (line 11). Next, the repository returns a list of artifacts. For each artifact, the algorithm looks for a sub-plan for such an artifact (lines 12-30). First, the context conditions are verified (line 13). If the context is satisfied (line 14), a new plan is created with the artifact (lines 15-16). If the list of dependencies of the artifact is empty (line 17), the new plan is added to the sub-plan (line 18). Else, if the artifact has dependencies, the algorithm is recursively called for these dependencies. If the result of the recursive call is not `NULL` (line 23), the resulting plan is added to the new plan and included into the sub-plan (lines 24-25). 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 19 and 26) and then the sub-plan is added to the resulting plan (line 32). Otherwise, if the conditions evaluation (line 13) returns `FALSE` or the recursive call returns `NULL`, the artifact can not be deployed. The loops continue for the other artifacts. If after all tries the sub-plan is `EMPTY` (line 34), the deployment for the selected goal is not possible, and the algorithm returns `NULL` (line 35). 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.

We should note that deployment plan is *valid* for a given context if: (i) for each artifact in the plan, all context conditions hold, (ii) for each dependency in a plan, there is at least one component within the plan that defines and one that implements the dependency. A deployment plan is *complete* if, for each goal in the request, there is at least one fulfilling artifact in the deployment plan. A deployment plan satisfies a deployment request if it is *valid*, and *complete*.

Being so, we can verify if a deployment plan satisfies a deployment request by executing the following steps that verify such properties: (i) check if for all selected artifacts, all context conditions are met; (ii) check if for all selected artifacts, the dependencies are within the deployment plan; (iii) 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.

- 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

Once the deployment plan has been devised in GoalD, the deployment execution will become ready to take place. The execution involves (i) *fetching* the artifacts present in the deployment plan from the repository to the target environment, and (ii) *binding* the components present into such artifacts, creating the application runtime architecture. The binding can then use alternative ways to bind between client objects to their requested implementation service at runtime. Using reflexive platforms allows eliminating

```
Input: DeploymentRequest request
Result: DeploymentPlan plan
var resultingPlan ← new DeploymentPlan()
foreach Goal selectedGoal in request.goals do
    var subPlan ← new DeploymentPlan()
    var definition ← repository.
    queryForDefinition(selectedGoal)
    if definition == NULL then
        return NULL
    end
    else
        subPlan.add(definition);
    end
    var artifacts ← repository.
    queryForImplementation(selectedGoal)
    foreach Artifact artifact in artifacts do
        var contextSatisfaction ←
        isSatisfied(artifact.conditions)
        if contextSatisfaction then
            var plan ← new DeploymentPlan ()
            plan.add(artifact)
            if artifact.depends == EMPTY then
                subPlan.add(plan)
                break
            end
            else
                var depPlan ← doPlanDeployment (artifact.depends)
                if depPlan != NULL then
                    plan.add(depPlan)
                    subPlan.add(plan)
                    break
                end
            end
        end
    end
    if subPlan != EMPTY then
        resultingPlan.add(subPlan)
    end
    else
        return NULL
    end
end
return resultingPlan
```

Algorithm 1: doPlanDeployment (Goals list)

static dependencies of components as available interface implementations are assembled at runtime.

To avoid static dependency between component implementations, the Dependency Injection[25] 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. [25] The basic idea of the Dependency Injection is to have a separate object, an *assembler*, that wires together the components at runtime[25]. 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.

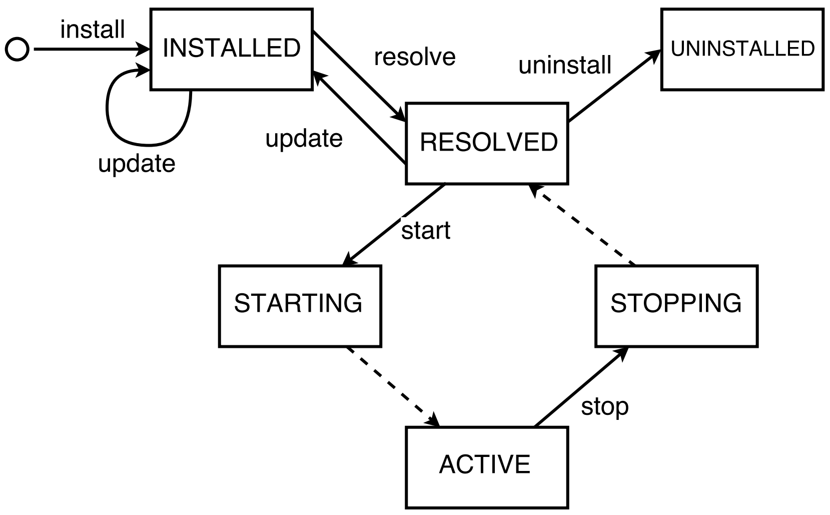


Figure 4.8: Representation of OSGi bundles lifecycle

In order to fetch and bind the components, we can make use of OSGi platform[59]. OSGi is a Java centric platform that allows dynamic fetching, binding and unbinding of components, usually named bundles. Figure 4.8 illustrates the lifecycle of bundles in the OSGi platform[59]. 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*. The integration to such approaches is seamless in our GoalD approach. Since the implementation of such integration is platform specific, it is left for a future stage of our work.

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: Are the offline activities of GoalD feasible to map artifacts from the CGM of the Filling Station Advisor case study?
  - Accurately maps goals, components and artifacts for the Filling Station Advisor case study.
- Q1.2: How long would the deployment plan algorithm take to deliver a result?
  - Time to produce a deployment plan.
- Q1.3: How reliable would a plan provided by the algorithm be?
  - Percentage of valid plan.

Our second goal G2 aims at providing a more comprehensible scalability evaluation of GoalD. The time required for planning the deployment could restrict the applicability of the approach. Some scenarios can require a timely response. Which could be the case for self-adaptable systems using adaptation at deployment to respond to changes in the computing environment or in user goals.

In such case, the number of artifacts needed to satisfy the deployment request as well as the number of variants of each artifact present in the repository could be high, putting pressure on the time needed to come up with a deployment plan. Hence, we define 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 various artifacts in the repository that provide the same goal but with different context conditions. We named the number of artifacts present in the repository that provide the same goal as variability level. The variability level can affect the scalability of the planning because it requires the algorithm to verify alternative dependency trees. This task can be computing intensive.

- Q2.2: How does the algorithm scale over the variability level of the repository?
  - M2.2: The time consumed to come up with a deployment plan, where each variability level accounts for a distinctive set of context conditions.

The experiments were conducted using a virtual machine in the Azure Cloud. A 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) was used. OpenJDK(1.9 64bits-build 9) 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 for the execution of the evaluation, the data obtained and scripts used to analyze it are available on a public repository[1].

5.1 Feasibility Assessment

We validated the feasibility of GoalD based on the Filling Station Advisor example.

Q1.1, mapping components and artifacts: We applied the patterns described in Section 4.1.2 to the case study specified in Figure 3.1. Then we defined the artifacts that would package that components following the approach proposed in Section 4.1.3. We then mapped 21 different artifacts.

Q1.2 and Q1.3: Planning time and Reliability: We instantiated an artifact repository with the mapped artifacts. We defined the following seven deployment scenarios with different contexts for the evaluation: (S1) simple phone with ODB2, (S2) smartphone with ODB2, (S3) smartphone without car connection, (S4) dash computer with GPS and no navigation system integration, and (S5) dash computer, connected, with GPS and navigation system integration. Scenarios (S6) dash computer without GPS and (S7) navigation system without Internet connection or available storage are scenarios for which there is no valid deployment plan. Figure 5.1 summarizes the computing heterogeneity that affects the system and the evaluation scenarios.

Q1.2: 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 times for each scenario. Table 5.1 shows the scenarios, the context, average time spent for planning in each scenario, in milliseconds together with standard deviation.

Q1.3: 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.

[1]Evaluation assets repository `https://github.com/lesunb/goald-evaluation/` last accessed on January 10th, 2017

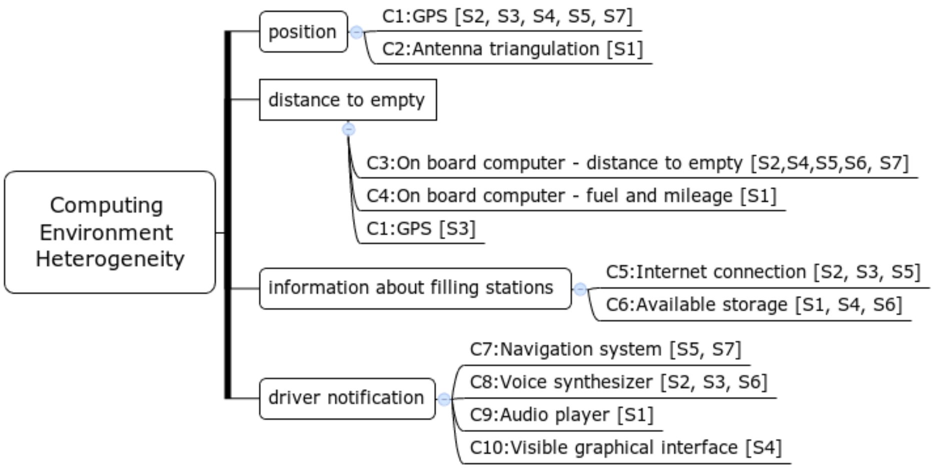
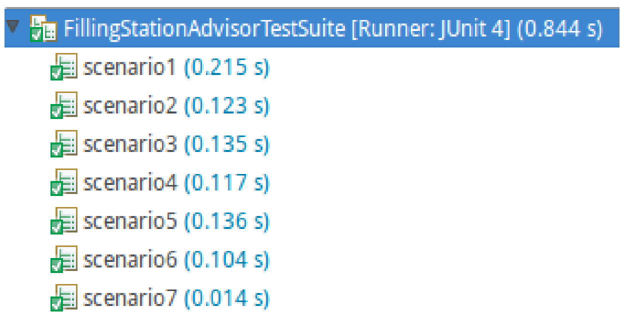

Computing Environment Heterogeneity
position
C1:GPS [S2, S3, S4, S5, S7]
C2:Antenna triangulation [S1]
distance to empty
C3:On board computer - distance to empty [S2,S4,S5,S6, S7]
C4:On board computer - fuel and mileage [S1]
C1:GPS [S3]
information about filling stations
C5:Internet connection [S2, S3, S5]
C6:Available storage [S1, S4, S6]
driver notification
C7:Navigation system [S5, S7]
C8:Voice synthesizer [S2, S3, S6]
C9:Audio player [S1]
C10:Visible graphical interface [S4]

Figure 5.1: Computing Environment Evaluation Scenarios

For scenarios S6 and S7, the algorithm returned `NULL` as expected, since in such scenarios we tested the case where there was at least one artifact in the repository not present for certain context conditions.

Table 5.1: Time to come up with a plan

Scenario	Context Condition	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



```
FillingStationAdvisorTestSuite [Runner: JUnit 4] (0.844 s)
    scenario1 (0.215 s)
    scenario2 (0.123 s)
    scenario3 (0.135 s)
    scenario4 (0.117 s)
    scenario5 (0.136 s)
    scenario6 (0.104 s)
    scenario7 (0.014 s)
```

Figure 5.2: Passing Tests

5.2 Scalability Assessment

Since the Filling Station Advisor has a limited size and does not allow for controlled experiments, we further evaluated our approach for scalability over the time to come up with a deployment plan. A repository as big as 140,000 artifacts was randomly generated. Out of the artifacts generated, different dependencies level of such artifacts was programatically configured for up until 3,000 hierarchical pathways to goals fulfillment. To empirically evaluate the impact of the various hierarchical levels on the deployment planning time, we executed 100 deployment planning requests and repeated each experiment 10 times.

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.

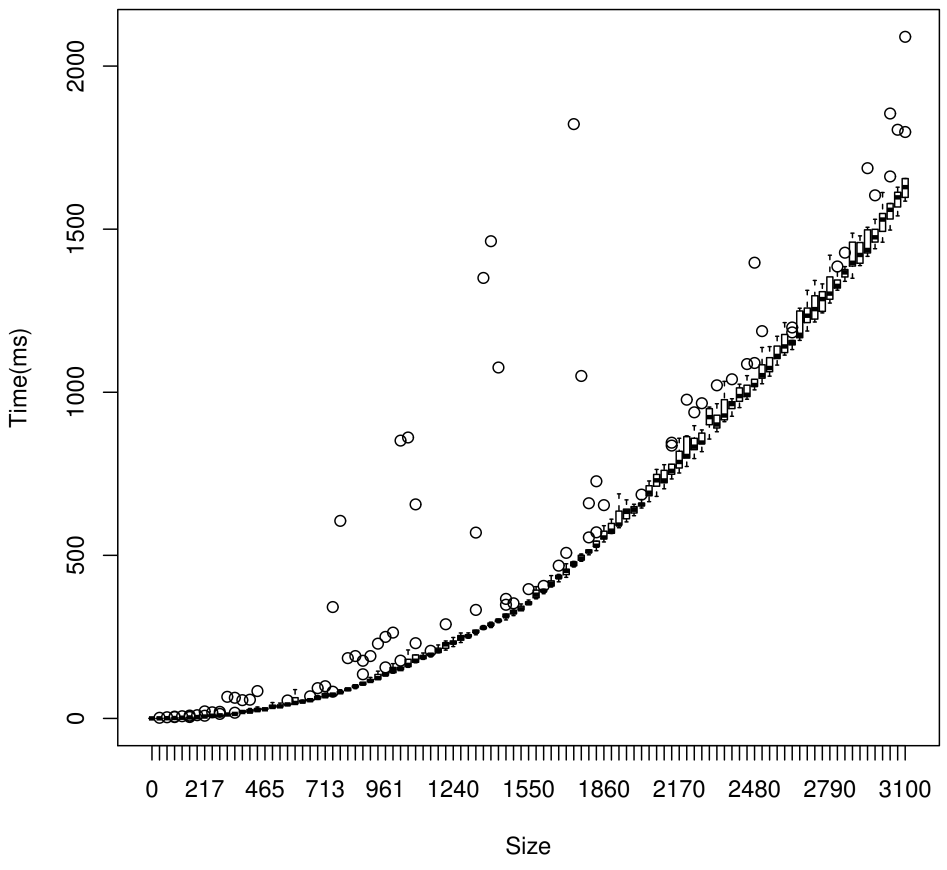


Figure 5.3: Scalability over the size of plan

Q2.2: How does the algorithm scale over the variability level on the repository? By answering this question we aim at evaluating how GoalD performs the planning for heterogeneous computing environments. For this purpose, we conducted the experiment for different levels of variability in the repository, from 1 to 10. By variability level we mean the number of different artifacts that implement the same goal, considering that each variability accounts for a distinctive set of context conditions. For example, for a variability level 2, there are two different artifacts that implement the same goal, where each artifact runs on a distinctive context condition.

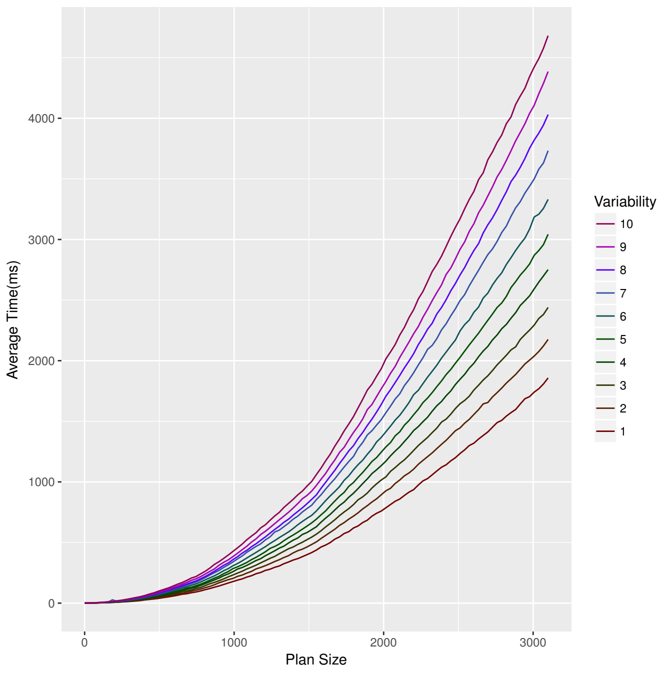


Figure 5.4: Scalability over variability level

The experiment was also repeated 10 times. We represent the average of the measures obtained for each experiment in Figure 5.4, where each curve represents a different level of variability. Results show that, in the worst case scenario where a deployment plan of 3,000 artifacts, with 10 alternative implementations for each artifact, it took less than 5 seconds to be planned by GoalD. Based on such results, the time spent to plan the deployment should not be an overhead to the deployment of the artifacts to the target environment, even on highly heterogeneous computing environments.

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 GoalD 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 section we discuss most related work whether they are goal oriented, handle heterogeneity, and support automated deployment.

Angelopoulos et al. [5] present an approach to handle variability at three different dimensions: goals, behavior, and architecture. Variability can occur at the goal level as an OR-refinement or context selection; at the behavior level as different plans flows; and at the architecture level addressing the variability of components and their implementations. GoalD can augment this approach by handling variability as a deployment problem and explicitly captures and caters for the different settings of the hosting environment.

Ali et al.[2] explore the optimization of the deployment for a given context variability space. Contextual Goal Models (CGM) are used to represent aspects of the environment elicited because of their relation to the solutions presented in the goal model. GoalD puts a primary focus on the context related to the computing environment and enriches the notion of resources and the mapping between goal achievement alternatives and software artifacts.

The Dynamic Software Product Line (DSPL) paradigm is motivated by a rapid production of software from a set of reusable assets to fit variability in users requirements and system environments. Bencomo et al. [12] use an SPL approach to adaptation by associating an architecture variability model with an environment variability model. Mizouni et al. [43] use a feature model associated with context requirements. The use of DSPL and its associated approaches is mainly focused on runtime adaptation where software systems switch amongst already implemented and deployed artifacts configurations. GoalD handles the step preceding that, i.e. the deployment stage and its decisions.

Leite et al. [37] 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. The abstract feature model is tailored to the variability of resources present in an inter-cloud environment, such as variability in processing power, memory, storage, and network. Different in GoalD the metamodel has no limit concerning the type of the resource. In Leite et al.'s the deployment adaptation is executed by solving variabilities in the deployment scripts. In GoalD we avoid the need for deployment scripts by putting the deployment variability at components level and providing a method to solve the variability autonomously.

Gunalp et al. [27] present an approach for automated deployment, which is the closest in nature to GoalD. In their approach, a specialist has to specify deployment a priori in terms of resources and their desired target states. They use an operational model to drive the adaptation: implemented strategies to move monitored resources to the target states. Differently, GoalD tackles the problem at the level of goals enabling an earlier treatment.

Chapter 7

Conclusion

7.1 Conclusion and future work

In this work, we presented GoalD, a novel approach to tackle deployment in highly heterogeneous computing environments. GoalD allows systems deployment to heterogeneous environments, partially unknown at design-time, without requiring a system administrator. We expect GoalD to be a suitable approach for scenarios that require an autonomous system, running on a heterogeneous environment, to interact with users and the physical world. This systems should use different resources to sense and actuate in the surroundings. This could be the case, for example, in a assisted living scenario, where the system could make use of different sensors to detect and emergency situation and also, could make use of different actuators to avoid further damages and call for help.

GoalD 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. GoalD 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 GoalD 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.

GoalD's goal model is based on Tropos syntax, which tree-like style reduce the complexity of analysis but also can be less expressive them others. The lack of expressiveness can lead to difficult at the design of the system. Which could limit the applicability of GoalD as it depends on the goal model to drive the design of the system at architecture and deployment levels. In future work, richer goal modeling syntaxes can be adopted to extend GoalD.

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[32] 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 GoalD with deployment planning for multiple nodes by including delegation as another form of variability; (2) evolve GoalD deployment planning in a self-adaptive approach for deployment, based on MAPE-K, with addition of monitoring, analyzing, and executing capabilities; (3) evaluate GoalD 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. *RE Journal*, 15(4):439–458, July 2010. 3, 9, 16

[2] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Requirements-driven Deployment. In *Software and Systems Modeling*, volume 13, pages 433–456, February 2014. 37

[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 LNCS, pages 51–75. Springer Berlin Heidelberg, 2013. vi, 7, 17

[4] Andersson, Jesper. A deployment system for pervasive computing. In *International Conference on Software Maintenance*, pages 262–270, 2000. 1

[5] Konstantinos Angelopoulos, Vítor E. Silva Souza, and John Mylopoulos. Capturing Variability in Adaptation Spaces: A Three-Peaks Approach. In *ER*, volume 9381 of *LNCS*, pages 384–398. Springer, 2015. 2, 18, 37

[6] Osamu Aoki. Debian Manual Chapter 2. Debian package management, 2016. 10

[7] Apache. Apache Maven Project, 2016. 10

[8] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer Networks*, 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. 31

[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] 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*, pages 23–32, 2008. 10, 37

[13] 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

[14] 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

[15] 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

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

[17] Chocolatey. Chocolatey - The package manager for Windows, 2016. 10

[18] Ivica Crnkovic and Magnus Larsson. Component-based software engineering-new paradigm of software development. *Invited talk and report, MIPRO*, pages 523–524, 2001. 19

[19] Ivica 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

[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

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

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

[26] D. Garlan, Shang-Wen Cheng, An-Cheng Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004. 11

[27] Ozan Gunalp, Clement Escoffier, and Philippe Lalanda. Rondo A Tool Suite for Continuous Deployment in Dynamic Environments. In *International Conference on Services Computing*, pages 720–727. IEEE , June 2015. 10, 38

[28] George T. Heineman and William T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 10, 11

[29] Homebrew. Homebrew, The missing package manager for macOS, 2016. 10

[30] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 1st edition, 2010. 9, 10

[31] 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

[32] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. 6, 7, 40

[33] Cornel Klein, Reiner Schmid, Christian Leuxner, Wassiou Sitou, and Bernd Spanfelner. A Survey of Context Adaptation in Autonomic Computing. pages 106–111. IEEE, March 2008. 6

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

[35] 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

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

[37] Alessandro Ferreira Leite, Vander Alves, Genaína Nunes Rodrigues, Claude Tadonki, Christine Eisenbeis, and Alba Cristina Magalhaes Alves de Melo. Automating Resource Selection and Configuration in Inter-clouds through a Software Product Line Method. In *8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, June 27 - July 2, 2015*, pages 726–733, 2015. 37

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

[39] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 3–14. ACM, 1996. 19

[40] 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

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

[42] 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

[43] 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

[44] 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

[45] Mirko Morandini, Frédéric Migeon, Marie Pierre Gleizes, Christine Maurel, Loris Penserini, and Anna Perini. A Goal-Oriented Approach for Modelling Self-organising MAS. In *ESAW*, volume 5881 of *LNCS*, pages 33–48. Springer, 2009. 8

[46] 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

[47] 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

[48] 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. *RE Journal*, 17(4):259–281, November 2012. 2, 12

[49] 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

[50] 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

[51] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive Software: Landscape and Research Challenges. volume 4, pages 14:1–14:42, May 2009. 6

[52] 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

[53] 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

[54] D. Spinellis. Don't Install Software by Hand. *IEEE Software*, 29(4):86–87, July 2012. 1

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

[56] Maxim Svistunov, Stephen Wadeley, and Tomáš Čapek. Red Hat Enterprise Linux 6 - Chapter 8. Yum, 2016. 10

[57] 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

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

[59] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.1.* 2007. 12, 30

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

[61] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, March 2000. 19

[62] 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

[63] 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