

Abstract

We see a growing interest in computing applications that should rely on a dynamic computing environment. This computing environment can be distributed, heterogeneous and composed of not reliable computing resources. These characteristics could potentially lead to a complex deployment process. Also, in such systems, for different reasons, it could not be desirable that the system is managed by humans. We proposed in this work a method for an autonomic deployment process that allows systems to deploy itself by reflecting about its goals and its computing environment.

Keywords: dependability, deployment, self-adaptive systems, autonomic systems, goal oriented requirements engineering



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

A Goal-Oriented Middleware for Dependable Self-Adaptive Systems

Gabriel Siqueira Rodrigues

Brasília
2015

Sumário

1	Introduction	iv
1.1	Introduction	iv
1.2	Problem	iv
1.3	Proposed Solution	v
2	Background	vi
2.1	Goal Modeling	vi
2.2	Context-aware Systems	vii
2.3	Software Components	viii
2.3.1	From Goals to Components	ix
2.3.2	Component-Based Adaptation	ix
2.3.3	Development of Self-Adaptive Systems	x
2.3.4	Software Deployment	xi
2.4	Multiagent systems (MAS)	xiii
2.5	Dependability	xiii
2.6	Attain Dependability at Runtime	xiv
3	Related Work	xv
3.1	Related Work	xv
4	Proposal	xvii
4.1	Proposed Method	xvii
4.1.1	Concepts	xvii
4.1.2	Off-line	xx
4.1.3	On-line	xxiii
4.2	Deployment Planning	xxix
	Referências	xxxi

Capítulo 1

Introduction

1.1 Introduction

As the frontier of computing system expands to more aspects of human life we get more systems that is meant to run in dynamic computing environment. Developing software for such environment is challenging [9].

Example of domains that leads towards dynamic environments are Ubiquitous Computing [10], Internet of Things (IoT)[6], Assisted Living[25], and Opportunistic Computing[37]. In such domains, the computing environment is dynamic as computer nodes can enter and leave the application. Also the computing environment can be highly heterogeneous as users have different sets of devices with different resources and characteristics. In addition, some characteristics of the computing environment can be unknown at design-time. For example, in a smart home application developer can know very little about the set of devices in the end-user home computing network.

Software deployment is the process of get a software ready to user in a given computing environment. Software deployment involves planning the deployment by assigning roles to a given computer node, selecting artifacts that must be present in the node and services that in must provide and consume. Also software deployment evolves copying artifacts for a target computing environment and setting appropriate configurations.

Traditionally software deployment is considered and human-centered activity only once executed. Such traditional view do not fit a dynamic computing environment, because in such environment the deployment should be adapted frequently. Previous works have explored goal-driven adaptation of dynamic systems at different aspects, such as configuration, behavior and structure[44]. Although, there is still a lack of proposals for platforms to handle autonomous deployment for dynamic environments.

1.2 Problem

In a dynamic computing environments the system is required to constantly deploy and adapt, as goals and resources change. This is opposite to the traditional view in which is expected that deployment is a human centered task executed only once. This work focus on the following challenges related to deployment in dynamic computing environments:

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

Challenge 2: uncertainty at design time. The system developer do not know the exact specification of the end user computing environment. This call for using abstract models of environment variability.

Challenge 3: dynamism. Nodes and resources can enter and leave the environment. In response, the system should constant adapt to keep hight level goals achievable despite changes in the environment.

Challenge 4: openness. Third party developers should be able to develop components to the system. The objective here is achieve decentralization and independence of provider. According, the system should not drive adaptation relying on models that can not be extensible at runtime.

Challenge 5: deployment specification accessible to users. Users may change their goals in a given environment and lead to the need to deployment change. A system administrator with knowledge in software deployment could be not available.

1.3 Proposed Solution

This work propose a method for autonomous deployment driven by context-goal model. Software centered systems that adopt this approach are able to accept goals, evaluate its capabilities and context and find the software modules that enable it to achieve its goals, fetch and install them. Also, the system can at runtime change its deployment in response to changes in its context, capabilities and available software modules.

The presented approach leverage context-goal models as the model that driven the adaptation. Goal modeling is an approach for system requirements that model the intentionality of actors. Context runtime goal models insert the context as another dimension, modeling the variability of interest in the environment as context an how it affect the system goals and means of achieving its goals. Using Context-goal models to driven the deployment, we can avoid rework by reusing a model already developed in a requirements eliciting stage. Also, because goal-models are highly abstract models, we can achieve a higher level of flexibility. In addition, by using user goals as a drive of adaptation we expect to make deployment configuration accessible to users, even if they do not have technical skills in system administration.

To execute the adaptation we propose the use of component-based adaptation in which the system is adapted by binding and unbinding software components at runtime. We find it promising as a component present a good level of abstraction (opposed to code or variable levels). It also builds upon mature component-based software engineering.

To allow open and decentralized evolution of the system we avoid the use of centralized design time models. Instead we propose break strategies to achieve goals as components that can be discovered at runtime. So third party developers can provide new components for achieve goals using different set of resources.

In order to easy the development of solutions that use the approach proposed in this work we are developing a reusable framework. The framework should have much the adaptation logic needed for the autonomous deployment.

Capítulo 2

Background

2.1 Goal Modeling

Goal Oriented Requirements Engineering (GORE) approaches have gained special attention as a technique to specify self-adaptive systems [32]. Goals capture, the various objectives the system under consideration should achieve.

Tropos[12] is a methodology for developing multi-agent systems that uses goal models for requirement analyzes. Tropos encompasses the software development phases, from Early Requirements to Implementation and Testing.

The Tropos key concepts

Tropos use a modeling framework based on i^* [43] which proposes the concepts of actor, goal, plan, resource and social dependency to model both the system-to-be and its organizational operating environment [12] [31].

Key concepts in the Tropos metamodel are:

Actor an entity that has strategic goals and intentionality

Agent physical manifestation of an actor.

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

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

Resource it represents a physical or an informational entity.

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

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

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

Contextual Goal Model

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

2.2 Context-aware Systems

Context-aware systems are ones that are able to adapt their behavior according to changing circumstances without user intervention.

[15] describe a framework for context-aware services.

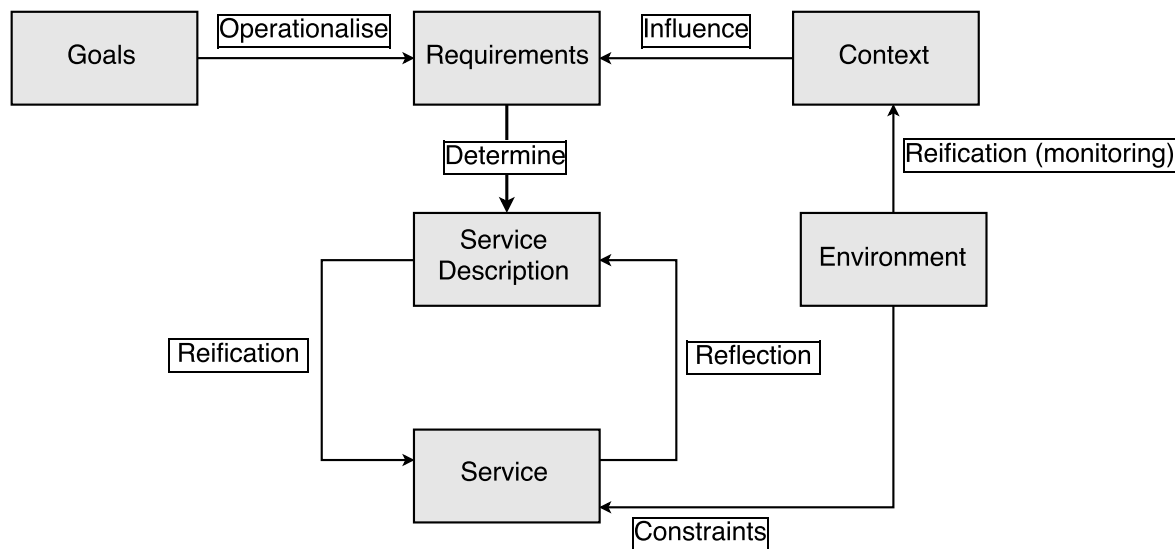


Figura 2.1: Context-aware services framework by [15]

Goal is an objective the system should achieve. It is an abstract and long term objective.

Environment is whatever in the world provides a surrounding in which the agent is supposed to operate. The environment comprise such things as characteristics of the device that the agent is supposed to operate in.

Context is the reification of the environment. The context provides a manageable, easily computer manipulable description of the environment. A context-aware system

should watch relevant environment properties and keep a runtime model that represents that information. By reasoning about that model the system can change its behavior. A context can be either a activator of goals or a precondition on the applicability of certain strategy to reach a goal.

A requirement operationalises a goal. It represents a more concrete, short-term objective that is directly achievable through actions performed by one or more agents.

Service description is the meta-level representation of the actual, real-world service. It should be a suitable formalism that allows services to be compared to requirements in order to identify runtime violations.

Service provides the actual behavior as perceived by the user.

The system should keep a causal connection between the service and the description. The system adapts by manipulating the service description.

2.3 Software Components

Heineman define *software component* as a “software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard”[22].

Software components is a unit of composition. Software systems are build by composing different components. Software components must conform to a component model by having contractually specified interfaces and explicit context dependencies only.[40].

Component based software engineering (CBSE) approach consists in building systems from components as reusable units and keeping component development separate from system development[13].

A *component interface* “defines a set of component functional properties, that is, a set of actions that’s understood by both the interface provider (the component) and user (other components, or other software that interacts with the provider)”[13]. 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.[22]

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[13].

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.[23]

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

Interface Components

Input, output

2.3.1 From Goals to Components

Lamsweerde [41] present a method for derive architecture from KAOS goal model. First an abstract draft is generated from functional goals. Secondly, the architecture is refined to meet non-functional requirements such as cohesion. the relation between software requirements and components.

Pimentel et al. [33] present a method using i* models to produce architectural models in Acme. If focus in he development of adaptive systems. First, it transforms a i* model into a modular i* model by means of horizontal transformation. Secondly, it creates an architecture model from the i* modularized model by means of vertical transformation. Architectural design models is made easier by the presence of actor and dependency concepts.

Yu et al. proposed an approach for keep the variability that exists in the goal model into the architecture. It present a method for creating a component-connector view from a goal model[44]. A preliminary component-connector view is generated from a goal model by creating an interface type for each goal. The interface name is directed derived from the goal name. Goals refinements result in implementation of components. If a goal is And-decomposed, the component has as many *requires* interfaces as subgoals.

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

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

```
Component G1 {  
    provides IG;  
}
```

```
Component G2 {  
    provides IG;  
}
```

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

2.3.2 Component-Based Adaptation

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

Rainbow[17] is a framework for self-adaptation architecture based. It keeps an model of the architecture of the system and can be extended with rules to analysis the system behavior at runtime, find adaptation strategies and perform this changes. It separate the functional code (internal mechanisms) from adaptation code (external mechanism) in a schema called external control, influenced by control theory.

MUSIC[35] project provides a component-based middleware for adaptation that propose to separate the self-adaptation from business logic and delegate adaptation logic to generic middleware. As in our propose it adapts by evaluating in runtime the utility of alternatives, to chose a feasible one (e.g., the one evaluated as with highest utility).

Dependency Injection

Dependency Injection is a pattern that allow for wiring together software components that was developed without the knowledge about it other. [16]

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

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

- impose compile time dependency between two classes
- impose runtime dependency between two classes

The basic idea of the Dependency Injection, is to have a separate object, an assembler, that wire together the components at runtime[16]. The client class refer to service using its Interface (the service interface). The assembler can use alternative ways of the *new* to instantiating an object, so that the wiring between client objects and implementation service classes could be postpone to runtime.

By this is we can at runtime:

- discover available implementation of a service interface
- decide this implementation to instantiate from available implementations

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

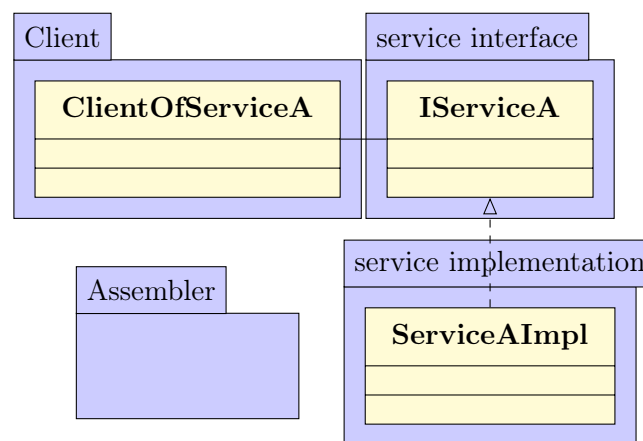


Figura 2.2: Two components

2.3.3 Development of Self-Adaptive Systems

For self-adaptive systems (SAS) some activities that traditionally occur at development-time are moved to runtime. In[3] was proposed a process for development of adaptive

systems. Activities performed externally are referred as *off-line activities*, and activities performed internally as *on-line activities*.

The right-hand side of Figure 2.3 depicts a running SAS. At this system we have *Domain Logic* that solves that is responsible for final user goals achievements. And also, *Adaptation Logic*, that is responsible to adapt the system in response to changes in the environment. Adaptation logic implements a control loop in line with the monitor-analyze-plan-execute (MAPE) loop [24].

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

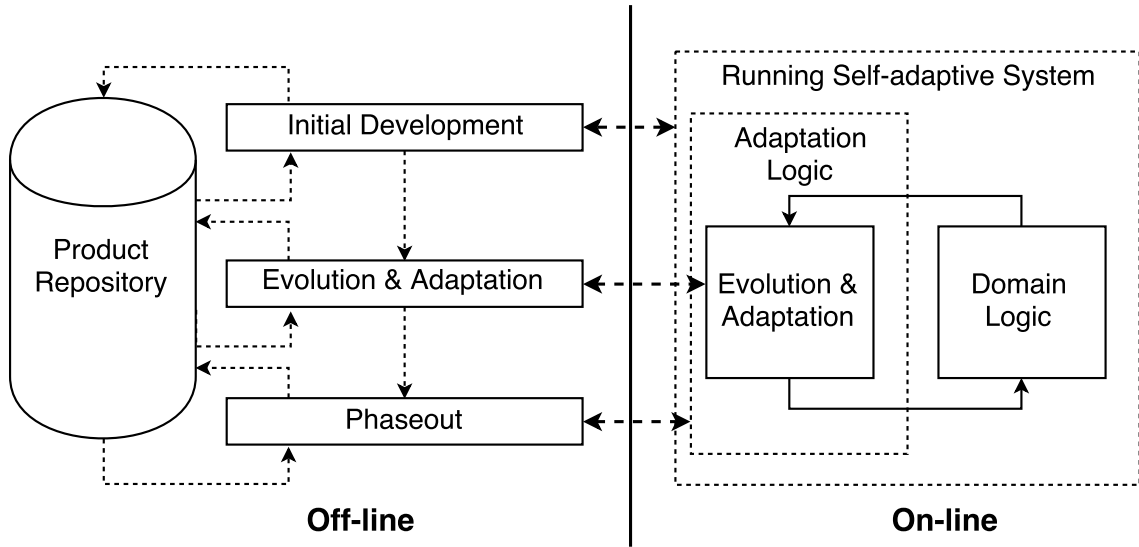


Figura 2.3: A Life-cycle model for Self-Adaptation Software System[3]

2.3.4 Software Deployment

Deployment is the process of get a software ready to user in a target computing environment. The deployment process can vary depending on the application domain and execution platform. In embedded platforms the deployment can consist in burn software into a chip. In consumer personal or business domain, for a desktop platform, the deployment can consist in a install process with collaboration between a person and a script that automate some steps. In a enterprise domain, for a web platform in can consist in coping and editing some files in a couple of machines. In many of this scenarios software will be periodically updated, frequently being unavailable in the process. The complexity of the software deployment can also vary in function of how much the platform is distributed (i.e. the number of nodes), how much heterogeneous it is, and how much is known about the deployment computing environment at design-time. In a dynamic and heterogeneous environment deployment can be specially complex.

Deployment artifacts are the artifacts needed at the deployment environment. Artifacts are build at development and build environment. Built artifacts are move for a delivery system where they can be accesses from the target environment. At deployment

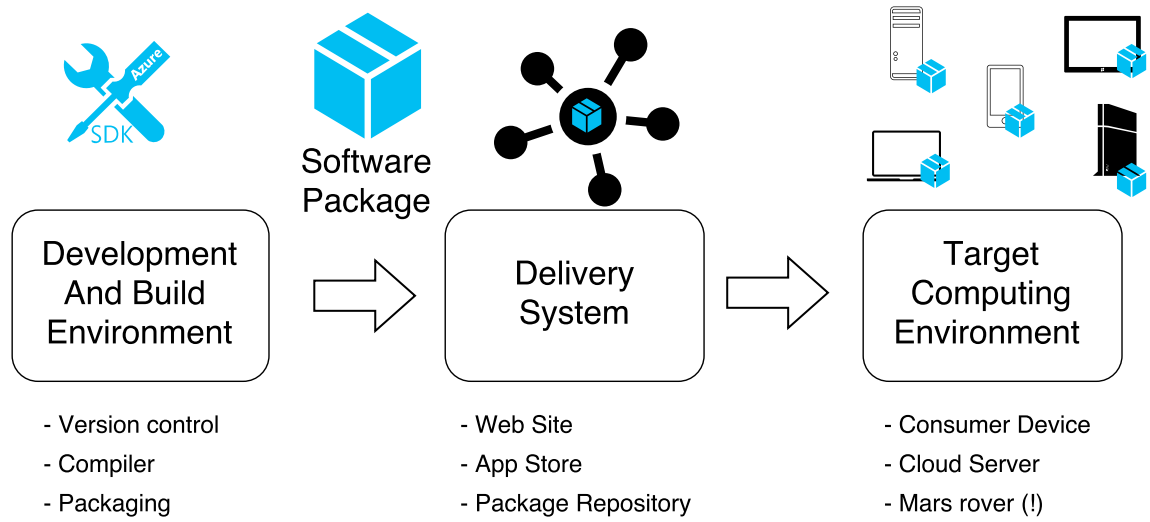


Figura 2.4: Artifacts Deployment

the artifacts are moved from the delivery system to the target computing environment. Also configuration activities can be realized. In the software industry, a *continuous integration* environment applies automation in building and getting components ready to delivery. In such a environment if a developer push changes to a code repository components are automatically built and published to delivery system. The build process commonly evolves fetching build dependencies, compiling source code, running automated quality control (tests and static analysis) and packaging. Artifacts are published if target quality policies are met. Fundamental to continuous integration environments are *Package Management Systems* tools. Theses tools simplify the process of manage software dependencies [38]. That tools ensure that development team members are working with same dependencies that are used in the build environment. *Continuous delivery* extends the continuous integration environment, moving components from the delivery system to a target computing environment with none or minimum human intervention. In modern enterprise environments the deployment activity has seen significant changes with popularization of use of virtual machines and cloud computing.

Devops[8] is a movement in software industry that advocates that all configuration steps needed to configure the computing environment should be written as code (*infrastructure as code*), and follow best practices of software development. That movement favor the documentation, reproducibility, automation and scalability. Devops allow for management of scalable computing environments. It can offer significant advantage for enterprise environment in relation to manual approaches in which system administrators configure the system by manually following configuration steps.

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

2.4 Multiagent systems (MAS)

Wooldridge [42] defines Multiagent Systems (MAS) as systems composed of multiple interacting computing elements known as *agents*. Agents are computer systems that are capable of autonomous action and interacting with other agents.

2.5 Dependability

Dependability can be defined as the ability of a system to avoid faults in its services that (1) are more frequent or (2) more severe than acceptable. Or as the characteristic of a system to be justifiably trusted.

A common terminology used for system deviations is the following: [7]

- **failure:** (or service failure) is a perceived deviation from the correct service provided by a system.
- **error:** is a deviation of correct internal system state that can lead to its subsequent failure.
- **fault:** is the adjudged or hypothesized cause of an error

Dependability includes the following attributes:[7]

- **availability:** readiness for correct service.
- **reliability:** continuity of correct service.
- **safety:** absence of catastrophic consequences on the user(s) and the environment.
- **integrity:** absence of improper system alterations.
- **maintainability:** ability to undergo modifications and repairs.

Many means have been developed of how to attain the attributes of dependability. These means can be classified as:

- **Fault prevention** means to prevent the occurrence or introduction of faults.
- **Fault tolerance** means to avoid service failures in the presence of faults.
- **Fault removal** means to reduce the number and severity of faults.
- **Fault forecasting** means to estimate the present number, the future incidence, and the likely consequences of faults.

Resilient systems [27] are expected to continuously provide justifiably be trusted services despite changes coming from the environment or from their specifications.

2.6 Attain Dependability at Runtime

To keep dependability in face of uncertainty in the deployment environment some techniques have been proposed for runtime analysis at runtime.

Felipe et al[19] propose a method of fault-tolerance for a scientific workflow execution in grid.

Alessandro Leite [14] propose a fault tolerance schema for cloud deployment based on which a fault instance in the cloud is monitored and in case of failure the instance can be restarted or terminated and then a new instance created.

Danilo et al[29] propose a methodology for fault forecasting by which developer, at design time, annotate the goal decomposition in goal model and specify context variables. A special tool generate a formula for, given a context, evaluate the probability of achieve a goal at runtime.

Capítulo 3

Related Work

3.1 Related Work

In the literature, there are proposals that tackle the problem of handling changes at system context [4][26]. These proposals focus on the system external context, reasoning about known facts about the external managed world and how it affects the systems goals. Asadie et al. [5] relates Goal-Oriented requirements and feature models for the development of SPL using a goal-oriented approach. In the described approach Goal-Models complement feature-models providing a mechanism to choose a set of features from stakeholder intentions. Angelopoulos et al. [4] present an approach to handle context variability at three different levels: goals, behavior and architecture.

Rainbow is a framework for self-adaptation architecture based[17]. It keeps an model of the architecture of the system and can be extended with rules to analysis the system behavior at runtime, find adaptation strategies and perform this changes. It separate the functional code (internal mechanisms) from adaptation code (external mechanism) in a schema called external control, influenced by control theory. [18] Different from our proposal it isn't goal-oriented and there is no work on how to relate Rainbow components to requirements.

MUSIC project provides a component-based middleware for adaptation that propose to separate the self-adaptation from business logic and delegate adaptation logic to generic middleware. As in our propose it adapts by evaluating in runtime the utility of alternatives, to chose a feasible one (e.g., the one evaluated as with highest utility)[35]. As Rainbow, MUSIC is not goal-oriented.

Salehie et al. [36] propose a run-time goal model and its related action selection. It models adaptable software as a system that exposes sensors and effectors and proposes a model consisting in Goals, Attributes and Action for selecting actions that will effect the adaptable software at runtime, giving sensed attributes. So the adaptation mechanism is to choose the best action given the actual attributes. As this work it uses explicit runtime goals and make them visible and traceable. Different from it we use a more symmetric approach that can allow for functional and adaptation management.

Güenalp et al. [21][20] propose a middleware for pervasive software with autonomic capabilities. The approach is service based. It proposes a component written in a custom language and the use of components repository that allows the discovery on new sensors. The system present a support for adaptability by using policies.

Tabela 3.1: Comparing characteristic properties of selected approaches related to Goald

Work by	Goal Oriented	Adaptive Deployment	Open Adaptation
Ali et al.[2]	Yes	No	No
Gunalp et al.[20]	No	Yes	No
Leite et al. [14]	No	Yes	No
Mizouni et al. [30]	No	Yes	No
Goald	Yes	Yes	Yes

Pinto et al. [34] introduces a approach to support traceability through requirements specifications, system architecture models, static and dynamic software design models and implementation artifacts of agent-oriented software systems. The authors use a set of types of relationships and structure the traceable information in levels (external, organizational and management) to improve the semantic of requirement traceability. The work also includes a process to be followed during the development of the traceability model

[11] uses a SPL approach. It associate a architecture variability model with a environment variability model. The environment variability is modeled as a transition system. The structural variability is responsible for the system adaptation. A configuration or a product is a set of component selected. A configuration is associated with states in the environment variability model. It focus on the adaptation in the configuration at runtime but not in the deployment. The variability model as presented do not allow for open adaptation. Mizouni at al. [30] uses a feature model associated with context requirements.

Ali et al.[2] explore the optimization of the deployment for a given context variability space in which the system will be deployed. It differ from this work in which we explore the variability in the computing environment itself, in terms of computing resources available to the system. We envisage that a joint usage with our approach would enrich the adaptiveness to the system to a a given environment.

In this work we present a Goal-Architecture view. Such view was presented in the literature. STREAM-A[33] create specialized agents. We create components. Components are not agents as they do not have autonomy, instead they are means of achieving a goal.

Leite et al. [14] proposes and 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 distributes across various clouds. The approach relies on centralized models of the environment and not address open adaptation problem.

Table 4.1 characterizes and compares research related to Goald.

Capítulo 4

Proposal

4.1 Proposed Method

An autonomous deployment platform should maximize the opportunity to make goals achievable by preparing the system for a broad range of context variability space.

This section describes our method to tackle the challenges presented in (1.2). It consists in offline activities for development of artifacts keeping trace to user goals. And in online support to autonomously handle deployment.

As offline activities, components are defined from a CGM using appropriate patterns. These components are packaged in artifacts together with metadata that describe what goals the packaged component can achieve, its context restrictions and dependencies.

At runtime the adaptation platform receives what goals the user wants to achieve in that environment and looks for artifacts that allow it to achieve its goals. The platform then assembles an architecture that allows the goal achievement.

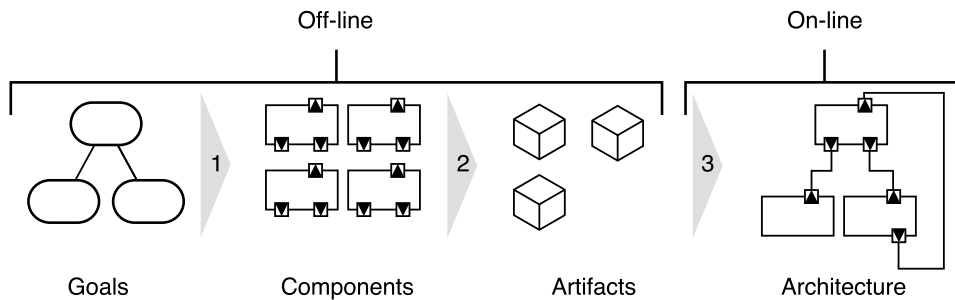


Figura 4.1: Transformations: (1) components definition; (2) packaging; (3) deployment

Relate Architecture with Requirements.

4.1.1 Concepts

Context and Context Conditions

In this work we are interested in conditions related with the computing environment. These conditions are related with availability of computing resources, e.g. take a game

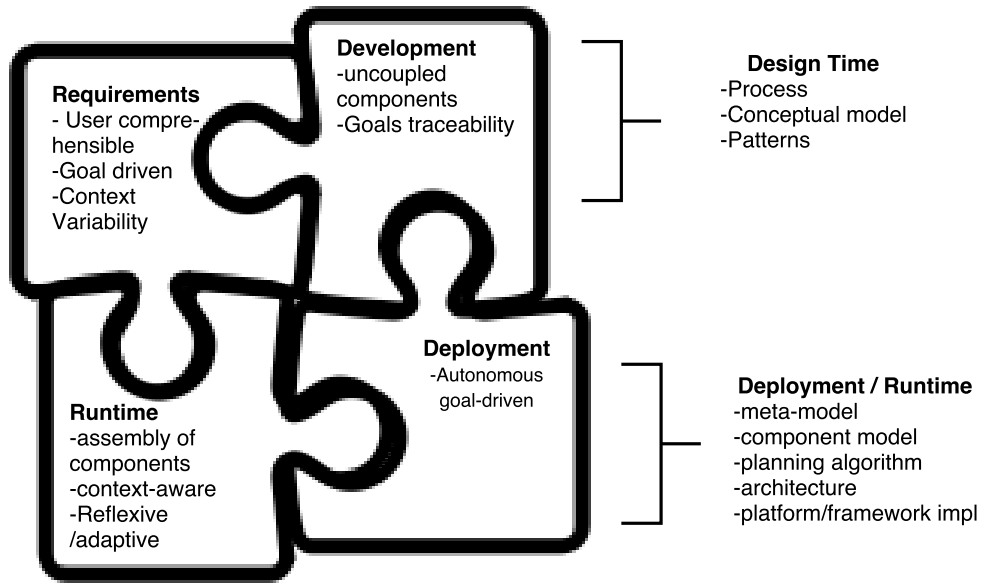


Figura 4.2: Proposal Overview

that can be rendered using CPU or GPU, the availability of GPU is a contextual condition that restrict the achievement of goal render game using GPU strategy. Adopting the context model proposed in [1] we will express such conditions as formulae in a set of facts. A given context satisfy a context condition if all related facts are monitored and the formulae associated are true.

The facts can be of an atomic proposition such as GPU, that is evaluated to true if an associated resource are known to be available, and it is evaluated to false otherwise. The facts can be also logical conditions using logical operators $==$, $!=$, $<$, $<=$, $>$, $>=$.

Goals, Components and Artifacts

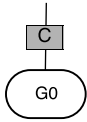
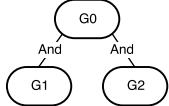
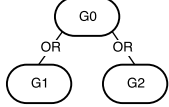
We enhance enhance component description with context condition. We extend Yu et al.[44] patterns for the Goals-Component view with contextual conditions.

Software component should be packaged in a standard packaging schema. The package should, also, have meta-data describing goals it provides, context conditions, and dependencies.

Artifact is a tuple (component, metadata) and metadata is a tuple (context conditions, dependencies, provided goals) context, goals and dependencies.

- Context conditions can be evaluated against the context. If the conditions are not satisfied the machine should not deploy the component.
- Dependencies are required services that should be provided by other components. Before deploy the component the machine should verify if it can satisfy the dependencies of the component.
- Provided goals are services that the component provide to the user or to other components.

Tabela 4.1: Contextual Goal Model to components; (1) context condition, (2) And-decomposition, (3)OR-decomposition

	<p>Component G0 { provides IG0; condition C; }</p>
	<p>Component G0 { provides IG0; requires IG1, IG2; }</p>
	<p>Component G1 { provides IG0; } Component G2 { provides IG0; }</p>

The artifact dependency: all that its components depend The artifact provide: all that its components provide The artifact conditions: all that its components conditions

If both context conditions and dependencies are satisfied, the artifact is capable of allow the goal achievement.

Online Goals

In traditional goal modeling the agent have a root goal that are refined in subgoals. Or-refinements allow for variability points in the goal model, but that refinement are static, at least for a given version of the goal model. We argue that in a dynamic and open environment that vision is not sufficient. We propose a different vision for the autonomous deployment. An agent should have a set of goals that it should pursue. That set can be updated by and user or the agent itself when following a self-assembly approach [39]. We call *active goal* a goal that the agent must pursue. In a dynamic environment, the agent can at a given moment have or not the resources to achieve a goal. *Achievable Goal* is a goal that the agent pursue and is capable of achieve.

In our deployment view of the goals, we will consider a goal achievable if we can deploy at least one artifact that provide that goals.

Evolution

evolve without great effort.

We argue that a goal model can be seen as a protocol definition. When we create OR-Refinements we are giving alternatives of execution, creating variability points.

By creating interfaces in variability points Open deployment platform.

and we create opportunity for thirty-parties provide new alternatives

having different context condition will allow the goal to be achievable in a broader range of contexts: for example, in screen controls will allow the game to be playable at touch screen devices.

In tradition deployment schema, a complete new version of the software should be released.

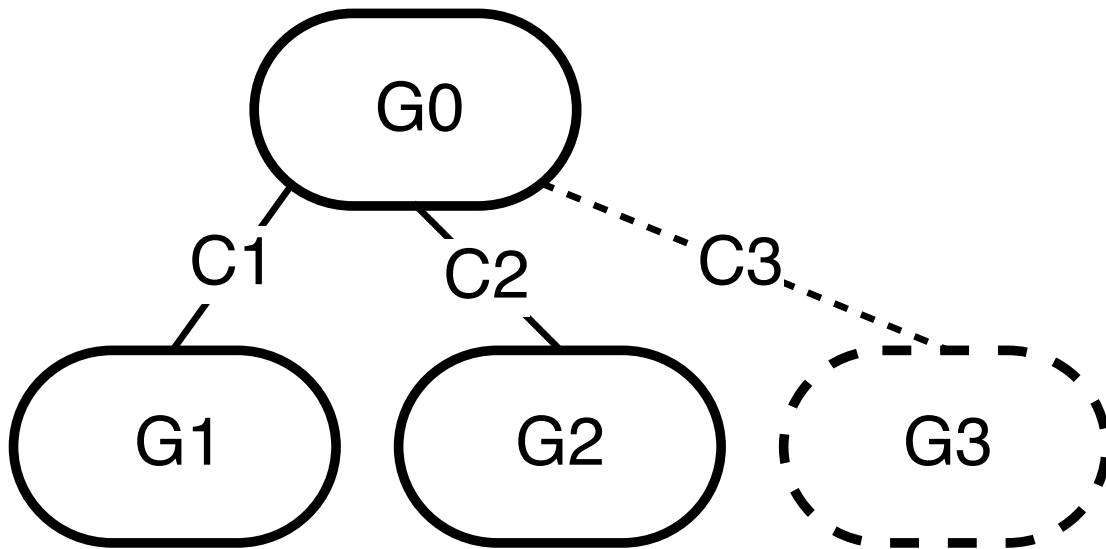


Figura 4.3: Evolution OR

Component development and release

Meta-model

4.1.2 Off-line

Roles

The proposed process considers three roles: users, requirements engineers and software architects. Figure 4.5 summarize the collaboration between the roles.

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

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

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

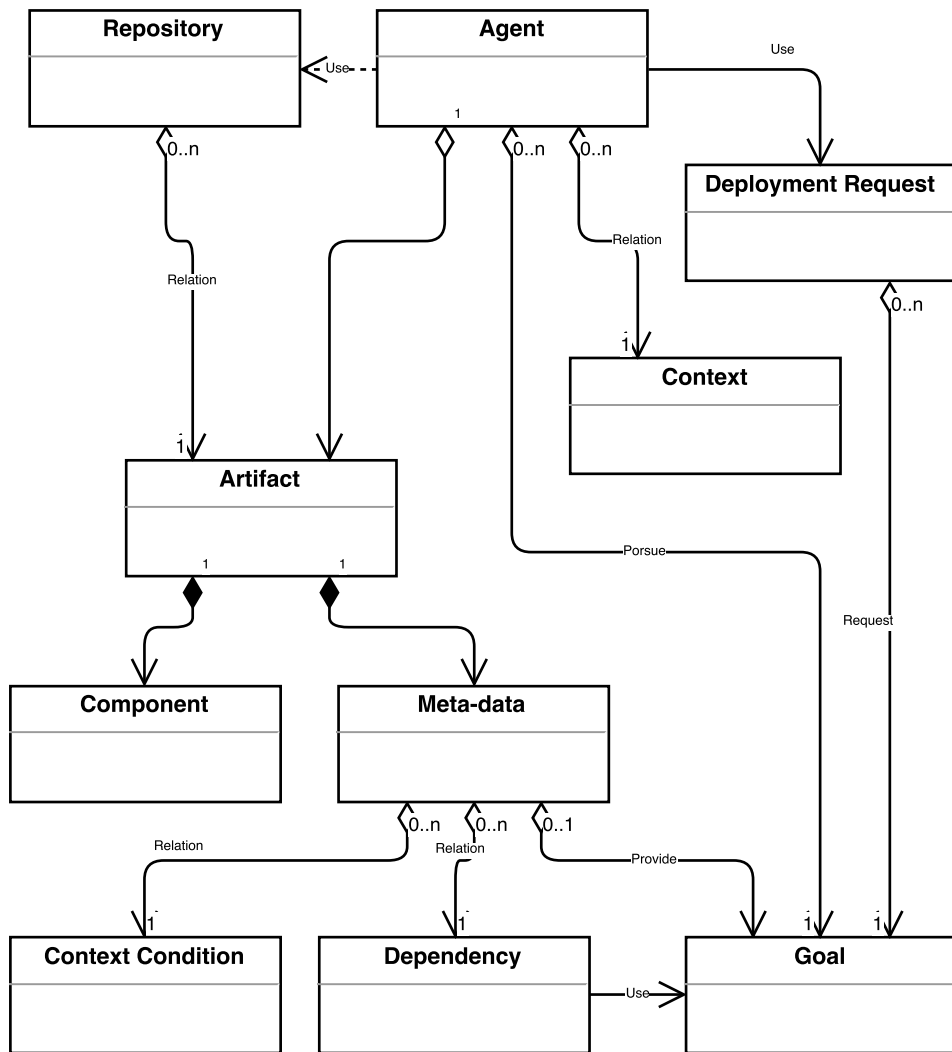


Figura 4.4: The Goalp Deployment meta-model

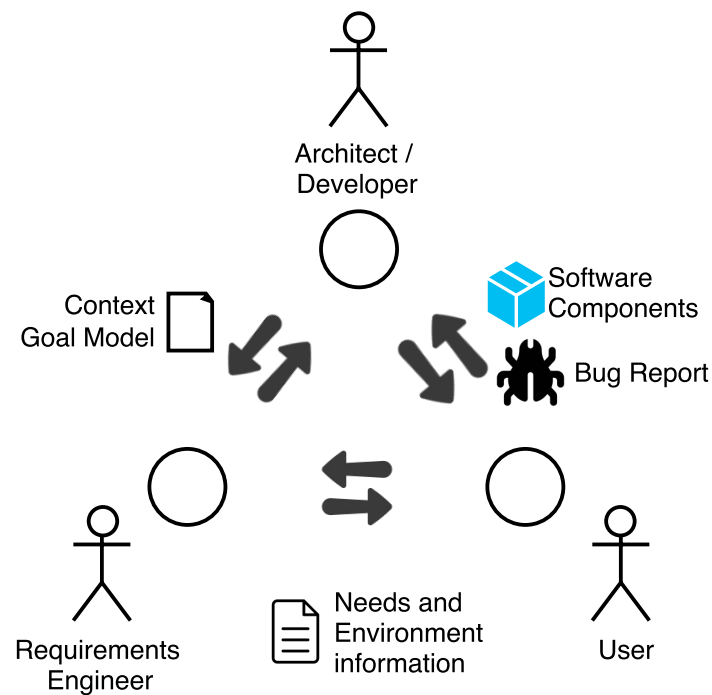


Figura 4.5: Roles collaboration

Activities

Figure 4.6 describe the development process activities.

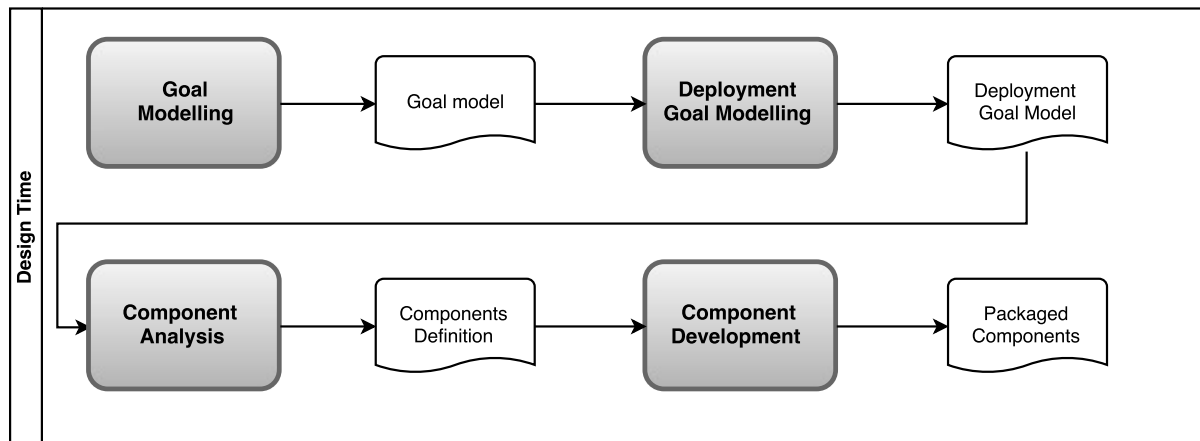


Figura 4.6: Deployment Process Activities

Goal Modeling

This phase is coordinated by a requirement engineer with participation of a domain specialist, possibly the user. A process such as TROPOS can be used. The output of this phase is a goal model. Here, the goal model assumes a central role in the software

development process. The goal model besides define the space of the solution, also works as common language. The goal model formalize the domain knowledge. It defines a common language between user and developers that will allow software deployment driven by user goals.

Context Goal Modeling

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

Component Analysis

Software engineer should identify variability points. Variability points in the contextual goal model are points where goals can be achieve with different strategies, each one having different context conditions. Component interfaces are created following the guidelines described in Section 4.1.1. The input and output of components are defined.

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

Component Development and Packaging

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

Deployment Actors

Repository

Node

Artifacts Interfaces for RC4 open adaptation Artifacts annotated with requirements and dependencies.

Figure Repository-Platform-Request

4.1.3 On-line

Deployment Description

Simple as possible to tackle RC5 (deployment specification accessible to users)

Goal Subgoals

Mandatory Goals Optional Goals

Example: Bomberman

PlayBomberman Mandatory:BombermanModel BombermanVideo BombermanControl Optional:BombermanAudio

Repo:BombermanControlKeyboard, BombermanControlJoystick

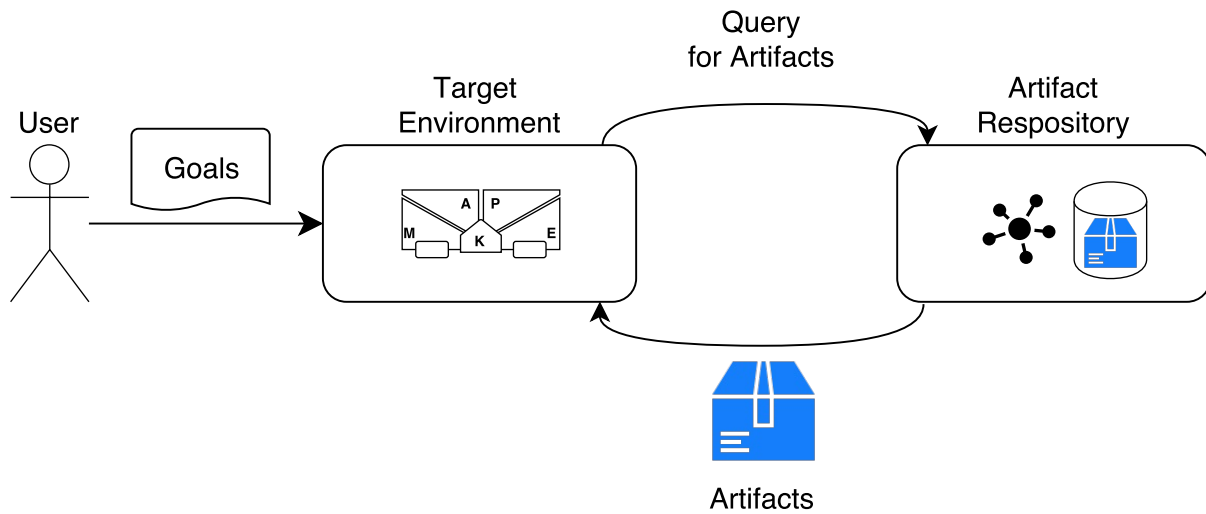


Figura 4.7: Goald Deployment Actors

Goald

Figure 4.7 depicts the deployment execution. A user interested in using a computing environment to achieve a set of goals submits to this environment which goals it wants to achieve in the form of a deployment request.

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

Facts about the computing environment are directed monitored by the agent by means of sensors. An Artifact should have in its meta-data the condition for its deployment. That condition is specified by a formula of facts.

Deployment Manager

Deployment Operations install uninstall

Component Model

- Requirements
- Dependencies
- Information it queries
- Listened Events
- Dispatched Events
- Periodic Execution

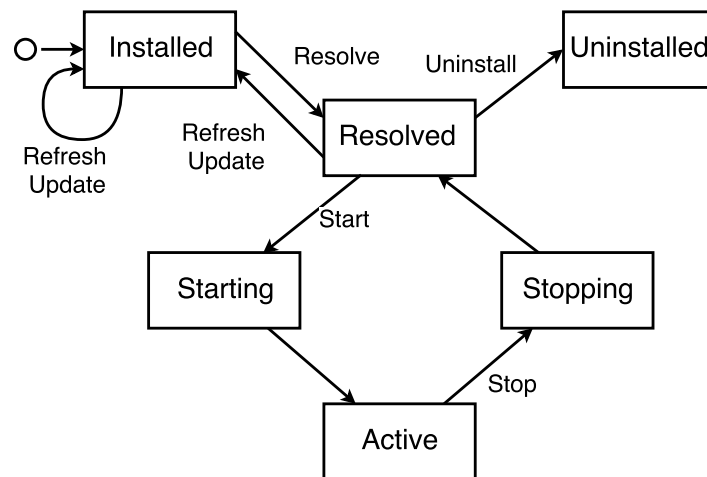


Figura 4.8: Component Life Cycle

Life cycle

- install
- register (listeners)
- uninstall

Dynamic binding

Application

Escalonando a aplicação e o código de adaptação.

Initial Setup

Knowledge base, Component model of the platform: mape-k is in it self a component.

Illustrative Scenarios

New Goal
Adapt to resource failure

Open Adaptation

Evolution 1: Notify user about not achievable goals

If a goal is not achievable a event is generated. With this another component can take action such as notify a external agent (e.g a user) that a goal is not achievable locally with current resources and known components.

Evolution 2: Adaptive monitoring policies

A monitor is associated with a monitoring policy that dictate the periodicity that the sense should occur.

- periódico:
- on demand: Is executed in response to a query
- listener: Is called by another system component or external actor.

Addressing Challenges

RC1 uncertainty at design time and RC2 heterogeneous computing environment

We address these challenges by assembling the system at runtime driven by user goals. Using our approach the developer do not need to know the exact specification of the user environment. We tackle challenge RC2 (heterogeneous computing environment) using decentralized approach to handle variability of computing environment.

RC3 dynamism We address this challenge by providing an adaptation framework to handle changes in the computing environment. Analyzing and responding to changes in the computing environment.

RC4 open adaptation We address this challenge by providing decentralized approach based on interfaces so that third party developers can provide new components to the system at runtime.

RC5 deployment specification accessible to users We address this by using goals as abstract way of specifying the system deployment. By this the user do not need to know details about system administration to configure a system. In our approach system administration rules and policies can be implemented as components.

Architecture

- events: are handled to registered listeners.

Knowledge Base

The knowledge base store a model of goals the system much achieve, the computing environment context and current deployed artifacts.

The information in the knowledge base can be of two types:

- facts: can be queried about by registered components

Monitors

Monitors are components that observes changes in the goals and context. All

- Goal Change Monitor is responsible for listening to request of include or remove goals.
- Computing Environment monitor is responsible to verify *facts* about the computing environment as introduced in 4.1.1. *Facts* are stored in the Knowledge Base and used to evaluate context conditions.

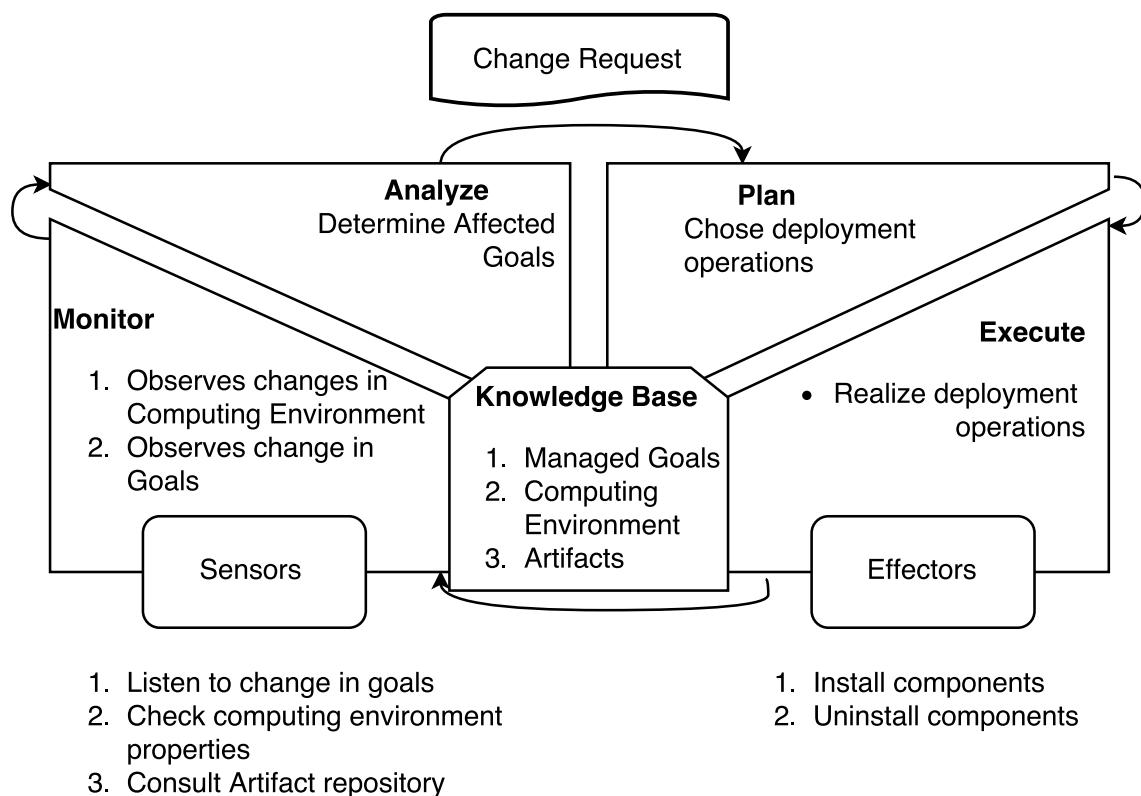


Figura 4.9: Goal Deployment Manager

- Repository monitors observe changes in software repositories.
- Introspective monitors observe the system state and behavior. (ex: monitor that check if there is any monitor that dispatches events that no one cares about)

The result of monitors can be:

- Change the knowledge base
- Dispatch Events

Core Components:

- Timer Monitoring Policy:
- Computing Environment Sensors: listen to timeouts of monitoring policy. Sense the environment. Update the knowledge base and dispatch events for changes in the environment.
- Goals change listeners: listen for external entities that want to change the goals of the machine. The interface is implementation dependent (e.g HTTP service, GUI, command line). Dispatch External Change Requests.
- Repository Monitor: queries repository for information about components.

Analyzer

Objective change analyzer

Computing environment change - evaluate if any system goal is affected.

Evaluate parametric formula for managed goals. If a goal probability of success drops below a threshold, dispatch deployment replanting event.

Handling evolution: simple approach favor a superior version.

- Goals Change- Check if a goal request is a change request. A goal removal will affect another goals?
- Adaptation In case of change in the available resources it should be analyzed if the change threatens the achievements of goals.

Listens to:

- changes in the environment
- external changes in goals

Dispatch

- Change Request: request a change in the deployment. Contains affected goals for what deployment should be replanned.

Planner

Receive change requests and enqueue it.

Deployment Change Planner is responsible for finding which operation should be executed in order to (1) make the active goals achievable. (2) Free up resource not associated with active goals.

How deployment is planned and the algorithm used will be described in Section 4.2.

Components

- Context Evaluation: it is responsible to evaluate if context conditions are satisfied for a given component in a given context.

Listens To:

- Change Request

Dispatch:

- Query Repositories: request information about available components that provide given goals.
- Execute Plan: request a deployment plan to be executed.

Execute

Executor components are responsible to actuate in the system. Deployment Change Executor is responsible for get components from repository and execute deployment operations such as install and uninstall components.

Listen To:

- Execute Plan:

4.2 Deployment Planning

Planning for new goals.

A goal is deployable if all its context conditions and dependencies are satisfiable. If a goal is satisfiable there is a deployment plan that is able to satisfy this goal. A deployment plan consist of a set of artifacts that form a closure in the dependency graph and all nodes has context conditions satisfiable.

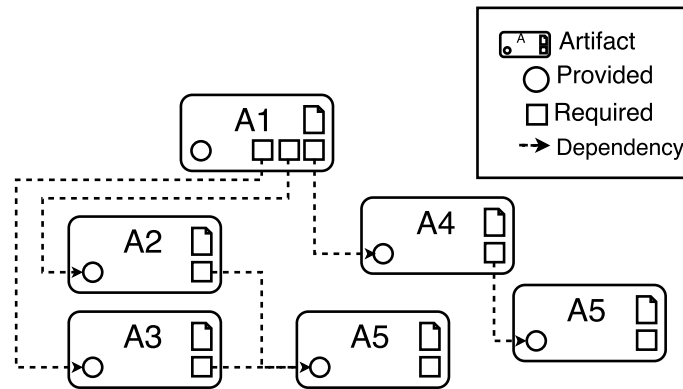


Figura 4.10: Dependency Graph

Algorithm 1: doPlanDeployment(Goal goal)

Input: A goal

Result: DeploymentPlan plan

```
1 List [] artifacts ← getArtifactsThatProvide(goal);
2 foreach Artifact artifact in artifacts do
3   Boolean contextSatisfaction ← isSatisfied(artifact.contextConditions);
4   if contextSatisfaction then
5     DeploymentPlan plan ← new DeploymentPlan();
6     foreach Goal dependency in artifact.dependencies do
7       DeploymentPlan subPlan ← doPlanDeployment(dependency);
8       if !dependencySatisfaction == NULL then
9         | return NULL
10      end
11      else
12        | plan.add(subPlan);
13      end
14    end
15    return plan
16  end
17 else
18   | return NULL
19 end
20 end
```

Referências

- [1] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, 15(4):439–458, July 2010. vii, xviii, xxiii
- [2] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Requirements-driven Deployment. *Softw. Syst. Model.*, 13(1):433–456, February 2014. xvi
- [3] Jesper Andersson, Luciano Baresi, Nelly Bencomo, Rogério de Lemos, Alessandra Gorla, Paola Inverardi, and Thomas Vogel. Software Engineering Processes for Self-Adaptive Systems. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, number 7475 in Lecture Notes in Computer Science, pages 51–75. Springer Berlin Heidelberg, 2013. x, xi
- [4] Konstantinos Angelopoulos, Vítor Souza, and John Mylopoulos. Capturing Variability in Adaptation Spaces: A Three-Peaks Approach. volume 9381 of *Lecture Notes in Computer Science*, Cham, 2015. Springer International Publishing. xv
- [5] Mohsen Asadi, Ebrahim Bagheri, Dragan Gasevic, and Marek Hatala. Goal-Oriented Requirements and Feature Modeling for Software Product Line Engineering. In *The 26th ACM Symposium on Applied Computing (SAC 2011)*, 2011. xv
- [6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Comput. Netw.*, 54(15):2787–2805, October 2010. iv
- [7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004. xiii
- [8] 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. xii
- [9] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward Open-World Software: Issue and Challenges. *Computer*, 39(10):36–43, October 2006. iv
- [10] Genevieve Bell and Paul Dourish. Yesterday’s Tomorrows: Notes on Ubiquitous Computing’s Dominant Vision. *Personal Ubiquitous Comput.*, 11(2):133–143, January 2007. iv

- [11] Nelly Bencomo, Peter Sawyer, Gordon S. Blair, and Paul Grace. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. In *SPLC (2)*, pages 23–32, 2008. xvi
- [12] 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. vi
- [13] Ilica Crnkovic, Judith Stafford, and Clemens Szyperski. Software Components beyond Programming: From Routines to Services. *IEEE Software*, 28(3):22–26, 2011. viii
- [14] Alessandro Ferreira Leite. *A user centered and autonomic multi-cloud architecture for high performance computing applications*. PhD thesis, Paris 11, 2014. xiv, xvi
- [15] Anthony Finkelstein and Andrea Savigni. A framework for requirements engineering for context-aware services. 2001. vii
- [16] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. x
- [17] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004. ix, xv
- [18] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. Software Architecture-Based Self-Adaptation. In Yan Zhang, Laurence Tianruo Yang, and Mieso K. Denko, editors, *Autonomic Computing and Networking*, pages 31–55. Springer US, 2009. xv
- [19] Felipe Pontes Guimaraes, Pedro Célestin, Daniel Macedo Batista, Genaína Nunes Rodrigues, and Alba Cristina Magalhaes Alves de Melo. A Framework for Adaptive Fault-Tolerant Execution of Workflows in the Grid: Empirical and Theoretical Analysis. *Journal of Grid Computing*, 12(1):127–151, October 2013. xiv
- [20] Ozan Gunalp, Clement Escoffier, and Philippe Lalanda. Rondo A Tool Suite for Continuous Deployment in Dynamic Environments. pages 720–727. IEEE, June 2015. xv, xvi
- [21] Ozan Günlalp, Levent Gürgen, Vincent Lestideau, and Philippe Lalanda. Autonomic Pervasive Applications Driven by Abstract Specifications. In *Proceedings of the 2012 International Workshop on Self-aware Internet of Things, Self-IoT '12*, pages 19–24, New York, NY, USA, 2012. ACM. xv
- [22] 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. viii
- [23] 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. viii

- [24] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. xi
- [25] Thomas Kleinberger, Martin Becker, Eric Ras, Andreas Holzinger, and Paul Müller. Ambient Intelligence in Assisted Living: Enable Elderly People to Handle Future Interfaces. In *Proceedings of the 4th International Conference on Universal Access in Human-computer Interaction: Ambient Interaction*, UAHCT’07, pages 103–112, Berlin, Heidelberg, 2007. Springer-Verlag. iv
- [26] Alessia Knauss, Daniela Damian, Xavier Franch, Angela Rook, Hausi A. Müller, and Alex Thomo. ACon: A learning-based approach to deal with uncertainty in contextual requirements at runtime. *Information and Software Technology*, 70:85–99, February 2016. xv
- [27] Jean-Claude Laprie. From dependability to resilience. In *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, pages G8–G9. Citeseer, 2008. xiii
- [28] 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. viii
- [29] Danilo Mendonça. Dependability Verification for Contextual/Runtime Goal Modelling, 2015. xiv
- [30] 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. xvi
- [31] 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. vi
- [32] Mirko Morandini, Frédéric Migeon, Marie-Pierre Gleizes, Christine Maurel, Loris Penserini, and Anna Perini. A Goal-Oriented Approach for Modelling Self-organising MAS. In Huib Aldewereld, Virginia Dignum, and Gauthier Picard, editors, *Engineering Societies in the Agents World X*, number 5881 in Lecture Notes in Computer Science, pages 33–48. Springer Berlin Heidelberg, November 2009. vi
- [33] João Pimentel, Márcia Lucena, Jaelson Castro, Carla Silva, Emanuel Santos, and Fernanda Alencar. Deriving software architectural models from requirements models for adaptive systems: the STREAM-A approach. *Requirements Engineering*, 17(4):259–281, November 2012. ix, xvi
- [34] Rosa Candida Pinto, Carla TLL Silva, and Jaelson Castro. A Process for Requirement Traceability in Agent Oriented Development. In *WER*, pages 221–232, 2005. xvi
- [35] 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. ix, xv

- [36] 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. xv
- [37] 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. iv
- [38] D. Spinellis. Package Management Systems. *Software, IEEE*, 29(2):84–86, March 2012. xii
- [39] 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. xix
- [40] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. viii
- [41] Axel Van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures*, pages 25–43. Springer, 2003. ix
- [42] Michael Woolridge and Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001. xiii
- [43] 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). vi
- [44] 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. iv, ix, xviii