Self-adaptation of Systems Composed by Adaptable Components

Liliana Wu Freitas Rosa
(Mestre)

Proposta de tese e plano de estudos para a obtenção do Grau de Doutor em Engenharia Informática e de Computadores

Júri

Presidente: Prof. Doutor João Manuel Pinheiro Cachopo
Orientador: Prof. Doutor Luís Eduardo Teixeira Rodrigues
Co-orientadora: Prof. Doutora Maria Antônia Bacelar da Costa Lopes
Vogais: Prof. Doutor José Alberto Cardoso e Cunha

May 7th, 2010
Abstract

This thesis proposal addresses the challenge of developing self-adaptive systems, when they are built from several adaptive components. The proposed approach to construct such systems is implemented for two case studies: micro protocol stacks and dynamic websites. This work contributions are several techniques that facilitate the design and implementation of such systems.
### Contents

1 **Introduction** 1  
  1.1 Problem Statement 2  
  1.2 Expected Contributions 2  
  1.3 Expected Results 3  
  1.4 Research History 3  
  1.5 Structure 4  

2 **Background & State of the Art** 5  
  2.1 Background 5  
    2.1.1 Key Concepts 5  
    2.1.2 Design Principles for Self-adaptive Systems 7  
  2.2 State of the Art 9  
    2.2.1 System Model 9  
    2.2.2 Models of Adaptive Behavior 11  
    2.2.3 Adaptation Support 13  
    2.2.4 Summary 16  

3 **Proposed Approach** 17  
  3.1 Reference Architecture 17  
    3.1.1 Models 18  
    3.1.2 The Offline Support 19  
    3.1.3 The Online Support 22  
  3.2 Case Studies 24  
    3.2.1 Case study 1: Micro Protocol Stacks 25  
    3.2.2 Case Study 2: Dynamic Websites 27  

4 **Work Progress** 29  
  4.1 Paper [RRL07a] 29  
  4.2 Paper [RRL07b] 30  
  4.3 Paper [RLR08] 30  
  4.4 Paper [RRL+09] 31  
  4.5 Ongoing work 32
Chapter 1

Introduction

Software is now pervasive in our daily life, and an increasingly number of tasks depends on the correct and efficient execution of programs that are built by composing multiple software components or services. These programs often need to operate in dynamic and unpredictable environments, and have to cope with events such as changes in the available resources, variable workloads, shifts in user requirements, etc. This motivates the design of self-adaptive systems, i.e., systems that continuously monitor the execution environment and, without human intervention, change their behavior at runtime in response to changes that may deviate their behavior or performance from the design goals.

However, the development of self-adaptive systems is far from trivial. In fact, it has been pointed out as one of the main challenges in Software Engineering [GvdHT09]. As mentioned in [CLG+09], the proper realization of self-adaptation functionality still remains a significant intellectual challenge due to the lack of methods, techniques, and tools that enable the systematic development of this class of systems.

The problem resides in the fact that the adaptation of complex systems, built as collections of components\textsuperscript{1}, is far more complex than adapting isolated components. Many systems already offer different options for customizing their behavior at runtime, including loadable modules and numerous configuration possibilities (e.g., Apache, Tomcat, MySQL, virtual machines). Such alternatives in individual services can be used to adapt the behavior of the composed system in response to changes in its execution environment. However, in the presence of several individual components that can be adapted in different manners, the management of the system adaptation becomes quite a complex task [Kep05]. The definition of an appropriate adaptation logic for the system, not only requires the detailed knowledge of every system component and its interactions, but also how it can be adapted and how that affects the remaining components. The space of possible configurations can easily become very large and, hence, finding the right adaptation configuration is quite difficult, time-consuming, and error-prone.

\textsuperscript{1}The term component is used to name components, services, protocols, or even technologies that offer a number of functionalities and serve a purpose in the system.
The research agenda presented in this thesis proposal aims to contribute with new techniques for developing self-adaptive systems built as compositions of adaptable components, that will help to cope with the inherent complexity of managing the system adaptation.

1.1 Problem Statement

The thesis addresses the engineering of self-adaptive systems built by individual adaptable components, which may have been developed by different teams in a non-coordinated manner. The aim of the work is to develop an approach to the construction of this class of self-adaptive systems and to contribute with a set of techniques that facilitate their design and implementation, addressing the challenges of adaptation management described previously.

The work will follow some key principles that have emerged from previous research [CLG+09, GSC09, OMT08], namely that mechanisms supporting self-adaptation should be separated from the actual system, and clearly identified in a feedback control loop. The key feature of the envisaged approach is the adoption of different types of adaptation policies ranging from event-condition-action rules to higher-level forms based on goals and utility functions.

To assess the proposed approach in different settings, the thesis will consider two different types of systems:

- Centralized systems composed of coarse grain components, namely dynamic websites, whose content is generated by multiple services.
- Distributed systems composed of fine grain components, namely modular communication protocol stacks composed of several micro-protocols.

By addressing two systems, it is expected to gain further insight on how to accommodate the differences among the systems while still using a common body of techniques to address the problem of self-adaptation. For instance, dynamic adaptation of distributed systems requires the use of coordination techniques that are not needed in centralized systems, but this should not prevent the use of similar methods for expressing the adaptation policies in both scenarios.

1.2 Expected Contributions

The main expected contributions are twofold, related to the specification of adaptation policies and with the mechanisms required to implement these policies.

On one side, the thesis will propose:

- modeling primitives providing means for expressing the adaptation logic of a system explicitly, separated from the description of the individual services, in two complementary forms:
1.3 Expected Results

The expected results of this work are:

- A engine to derive at runtime the adaptations, from high-level policies that combines input from the structural representation of the system, the characteristics of each individual component and available adaptations, and context information regarding the execution environment.
- The design of RAppia, a reconfigurable version of the Appia protocols composition and execution framework.
- The design of a set of context sensors and actuators to support the runtime adaptation of dynamic websites.
- A prototype and an experimental evaluation of an adaptable group communication protocol.
- A prototype and an experimental evaluation of an adaptable dynamic website.

1.4 Research History

This work has background in my Master dissertation, where I studied a policy-oriented approach to the construction of adaptive communication protocols [RLR06]. The adaptation policy is a collection of low-level event-condition-action rules, specifying which adaptations are executed for each relevant change in the context. From this work experience, it was possible to make the following observations:
In a distributed setting it is necessary to identify, not only what needed to be adapted, but also which coordination strategy should be used. This motivated us to study ways to automate the selection of the coordination strategy.

The previous work quickly led to the observation that it would be interesting to automate the generation of even a larger portion of the adaptation policy, ideally requiring the user to express the policy using high-level goals.

Furthermore, the doubt of whether the techniques initially developed for communication systems composed by micro-protocols could also be applied to the composition of other, more general, and potentially coarser grain, components has also influenced this thesis proposal. My visit to AT&T Labs Research, and the collaboration with Matti Hiltunen and Richard Schlichting was instrumental to expand these ideas to the web.

Eventually, the idea of automatically extracting the policy was first experimented in the dynamic website context and only later applied to the composition of distributed micro protocols.

It is worth to mention that a number of works related with this thesis have been developed in the GSD group at INESC-ID during my PhD program. Cristina Fonseca did a study of the advantages of using specialized switching protocols to speed up the coordination required to perform distributed reconfiguration [FRR09]. João Ferreira applied some of these ideas in the design of A-OSGi, an adaptable OSGi framework [FLR10]. Tiago Taveira will work on the implementation of the reconfigurable group communication system.

1.5 Structure

The thesis proposal continues in Chapter 2 with a description of this work background and the state of the art. This is followed by Chapter 3 where the proposed approach is outlined, giving an overview of the techniques and their support. The chapter also includes the approach evaluation, namely, the case studies that will be used. Chapter 4 gives insight on the work progress, establishes the a planning for the future work, and Chapter ?? concludes the thesis proposal.
Chapter 2

Background & State of the Art

This chapter starts by reviewing a number of key concepts and design principles related to adaptive systems. Next, it presents an overview of the state of the art. The analysis of the state of the art will allow to identify a number of challenges and issues posed to the design and implementation of composed adaptive systems. These challenges provide the motivation for the proposed approach.

2.1 Background

This section addresses the nuclear concepts employed in the research area of the thesis. The literature includes sources from different research areas but the majority are from renowned conferences on Autonomic Computing, namely the International Conference in Autonomic Computing (ICAC), and on Software Engineering, namely the International Conference on Software Engineering (ICSE).

2.1.1 Key Concepts

The term adaptation is widely used in several domains, with slightly different meanings. In general, software adaptation concerns changing a software system with a purpose, in response to variations in its operational envelope, for instance, in the user needs, in the system workload or in the available resources. Dynamic software adaptation (also known as runtime software adaptation) imply that the intended changes take place during the system execution [MSKC04b]; there is no need to stop the system.

Self-adaptive systems, or simply adaptive systems, are systems that are able to adjust their behavior autonomously (i.e., without human intervention), in response to their perception of the environment and the system itself for a variety of goals [CLG+09]. Self-adaptive systems are also referred by more specific terms when self-adaptation exclusively addresses a specific type of goal. For instance if adaptation concerns dependability or performance, systems are said to be
self-healing or self-optimizing. Systems or components are said to be adaptable if they provide mechanisms for being adapted during runtime but they do not monitor the environment and the control over the change in its behavior is left to other components.

Self-adaptive systems are sometimes referred to as autonomic systems. This is because self-adaptation is a means that has been explored to in the context of autonomic computing — a vision of the future in which software systems will manage themselves in accordance with high-level objectives specified by humans [Kep05]. This self-management relies in four concerns: self-healing, self-protection, self-optimization, and self-configuration.

Two key elements in self-adaptive systems are the context and the adaptation logic. The former is the information regarding the execution environment, which has to be monitored due to its relevance to assess if the system behavior is suitable or has to be changed. The latter concerns the decisions that determine when to adapt and how.

2.1.1.1 Context

The notion of context and context-awareness was first introduced in computer science to improve the interaction between humans and computers. Context provides implicit situational information, as regular communication between humans would have and was lacking in human-computer interaction [Dey01]. In a broad sense, context can be understood as any information that characterizes the situation of an entity [Dey01]. In adaptive systems, this boils down to any information that can be used to detect the situations in which the system needs to adapt. For instance, context might include information that allows to evaluate if the system meets certain quality of service requirements (performance, availability, security, etc).

Context information can have different origins. Usually, it encompasses both fine-grained and coarse-grained information. The gathering of context information can be done in different ways: periodically, continuously, or on demand. Hence, key aspects in the development of context-aware systems [SAW94] are: (i) the identification of context sources, (ii) acquisition of information from sources, (iii) aggregation of context information, and (iv) analysis and publication of the analysis results.

2.1.1.2 Adaptation Logic

The adaptation logic of an adaptive system is what defines its adaptive behavior. This includes, on the one hand, the characterization of the situations in which changes are needed and, on the other hand, the definition of what type of adaptation is required in each of these situations. In component-based software systems, adaptation is typically classified as behavioral adaptation or structural adaptation [MSKC04a]. Behavioral adaptation takes place if the system dynamically changes its behavior without changing its structure. For instance, adjusts the value of some parameters to tune the system behavior or change some variable that determines the algorithm that is used. Structural adaptation occurs if the system’s structure is modified at
runtime (for instance, a software component is replaced with another component that has the same interface or, more generally, components and connectors are arbitrarily added or removed). While the variability supported by behavioral adaptation is fixed (in the sense that the number of variants becomes fixed at time of construction of the system), structural adaptation supports an unbounded number of variants because adaptation may introduce an element made available subsequently.

### 2.1.2 Design Principles for Self-adaptive Systems

Self-adaptation can be handled within the system, at the code level, for instance making use of programming language features such as exceptions, reflection, and aspects. However, the complexity and cost of developing and maintaining adaptive systems can become extremely high when ad-hoc solutions, defined on a per-system basis, are adopted. Considerable research has been done in the area of dynamic adaptation and self-adaptation in order to establish approaches that allow the construction and maintenance of adaptive systems in a cost-effective way. Among the key design principles that have emerged from this research, the most significant are: (i) the separation of the concerns of system functionality from the concerns of self-adaptation and (ii) explicit representation of a feedback control loop.

#### 2.1.2.1 Separating Adaptive Behavior

When the system’s adaptation logic is scattered by several components, it becomes challenging to understand the outcome of adapting in particular scenario, and reuse the components or the adaptations. Altogether, this scattering makes it hard to maintain and costly to modify.

The separation of concerns (SoC) paradigm [HL95] is a widely accepted solution to these issues, allowing a cost-effective construction of self-adaptive systems. The solution requires that all aspects that concern adaptation are extracted from the base system and treated separately from the system [CGS05]. To some extent, self-adaptation is made external to the base system, which, for this reason, is also referred to as the managed system. By applying this principle it is possible to: (i) decrease the complexity of the software development, (ii) facilitate comprehension, and (iii) promote reuse of adaptation solutions.

#### 2.1.2.2 Closed-loop Control System

A number of recently developed approaches implement the separation and externalization of the adaptive behavior in terms of a control layer on top of the managed system. This layer monitors the base system, possibly maintaining an explicit model of the system, and, relying in a set of high-level goals, adapts the behavior or structure of the system. In many approaches it requires the insertion of probes in the base system (for instance, to detect specific system events) and effectors that can perform a specific set of adjustments. This technique can be applied to
both recently built or legacy systems, and further facilitates the reuse across different systems, reducing the cost of developing new self-adaptive systems.

Many approaches rely in a particularly popular external control mechanism reminiscent from the classical control theory: the closed-loop control [SEM03], depicted in Figure 2.1. In a closed-loop control system, a sensor monitors the output and feeds the data to a controller which continuously adjusts the input as necessary to keep the error to a minimum or to maintain a goal. Feedback on how the system is actually performing allows the controller to dynamically compensate for disturbances to the system. An ideal feedback control system cancels out all errors.

![Figure 2.1: Closed-loop control system](image)

In adaptive systems, the control loop consists of mechanisms that monitor the system, reflect on observations for problems (check if the observed behavior meets the system requirements), and control the system to maintain it within acceptable bounds of behavior [SEM03]. Furthermore, to reflect on observations, the external control system requires an explicit model of the system being maintained, to be used as a basis for adapting the system [OGT+99].

A principle that has been strongly advocated is that control loops are elevated to a first-class entities in the development of self-adaptive systems [CLG+09]. That is to say, the feedback control loop needs to be an explicit and visible element of the system in its modeling, design, and implementation.

In adaptive systems, a number of variations to the feedback control system are used. Despite a coarser or finer separation of aspects, it is possible to identify several activities. The following description presents the separation of activities in the perspective of autonomic computing [KC03b], as depicted in Figure 2.2. There are four activities. The monitoring activity is responsible for gathering the context information regarding the execution environment, the system itself, and any other information considered relevant for adaptation. This is done relying in sensors. The analysis activity analyzes and interprets the collected information according to the system model, in order to detect deviations from the desired system behavior. The planning activity uses the information provided by the monitoring and analysis activities to decide which actions must be performed to return the system back to an acceptable state. Finally, the execution activity performs the system adaptation, applying the actions decided in the previous phase through effectors. The division in several activities enables the independent modification
and extension of each one.

![Autonomic manager control loop](image)

Overall, employing a feedback control system as an external control mechanism to support adaptation results in an infrastructure that is highly reusable and easy to modify, whenever changes to the adaptation support are necessary. These characteristics are paramount when designing and implementing adaptive systems.

### 2.2 State of the Art

This section gives an overview of the state of the art for adaptive systems. Due to the sheer volume of work that employs adaptation and the expected size of the thesis proposal, this overview will mainly focus on approaches in which self-adaptation is made external in a control layer, separated from the other concerns, and relies on an explicit feedback control system.

In this class of approaches there are several critical design issues. One is the choice of type of system model. The system model is used by the external feedback control system to detect changes in the context and select adaptation strategies. Another is the choice of model for the adaptive behavior, which allow to express the adaptation logic. Finally, the last critical choice is the type of adaptation support offered to the control layer, responsible for executing adaptation and providing updated information on the system and execution environment.

#### 2.2.1 System Model

As mentioned previously, in approaches to self-adaptation that rely on closed-loop control, systems maintain a model of themselves, useful to provide information to detect changes in the behavior, but also indispensable to help select an adequate adaptation strategy. The choice of model type is intrinsically connected with the type of adaptations that are expected to be supported. For example, a component-based system may be adapted by adding or removing components, and by changing links between components. To perform such adaptations, the system model must capture the components that are currently in use in the system and how
CHAPTER 2. BACKGROUND & STATE OF THE ART

you are connected.

This type of system model is known as an architectural model: it represents an abstract view of the system as a composition of computational elements and their interconnections [SG96]. Architecture-based approaches to self-adaptation rely in this type of system model. Adaptation actions in these approaches are typically restricted to operations that can change the structure of the system architecture: additions, removals, replacements, and (dis)connection of architectural elements. When behavioral adaptation is needed, for instance, to tune core parameters of the system, the system model must capture which parameters are available and their current value.

Taylor and colleagues [OMT98, OGT+99] demonstrated the beneficial role of an explicit architectural model fielded with the system and used as the basis for runtime change. In a chapter devoted to self-managed systems in Future of Software Engineering 2007, Kramer and Magee also state that architectural models seem to provide the required level of abstraction and generality to deal with the challenges posed by self-adaptation [KM07]. In the last years, several architectural-based approaches to self-adaptation have been proposed. Some more prominent examples are briefly described below.

- **Rainbow framework** [GCH+04] — it offers structural self-adaptation, maintaining an architectural model of the system and mechanisms to update the model according to changes to the system. The approach relies in a mapping between model elements and system-level elements. A detailed description of this framework is given in Section 2.2.3.2.

- **Mobility and Adaptation Enabling Middleware (MADAM)** [FHS+06] — a reference architecture and middleware for mobile computing applications that supports flexible context monitoring, adaptation planning and dynamic reconfiguration through the change between different variants for the same variation point.

- **Three-layer reference model** [KM07, SHMK08] — a reference architecture that relies in a three-layer model for adaptable software architectures and task synthesis from high-level goals. The top layer that generates plans from high-level goals. The middle layer that constructs component configurations from plans and executes those configurations. Finally, the bottom layer includes the components implementation.

- **Policy-based architectural management (PBAAM)** [GT09] — an approach to self-adaptation that uses architectural models as runtime artifacts, adopts a policy-based model of adaptive behavior that can be modified at runtime and provides explicit support for the recording and visualization of adaptations in order to help the operators and designers.

\[1\]The first work was considered the most influential paper of ICSE98.

\[2\]A plan specifies which actions will lead the current state to the goal state, in the form of condition-action rules.
2.2. Models of Adaptive Behavior

In approaches to self-adaptation that separate the adaptation behavior from non-adaptive behavior, an important decision is the choice of the type of model of adaptive behavior that is considered. Approaches that focus on structural adaptation often opt for graph-based models. In this case, adaptation boils down to graph rewriting, and can be expressed, for instance, using graph transformation rules [BLMT08, TGM99, HIM00], graph grammars [LM98], or programming scripts for graph manipulation [WF02, GCH04] (also called repair strategies). There is also a number of approaches that use process-based models [ADG98, Oqu04] and others that use state-based models [ZC06, DHPB03]. For instance, the AutoTune [DHPB03] agent framework employs state-based models to uniquely adapt a set of system parameters. These models are obtained from equations that relate system properties and parameters. The equations are determined in an experimental manner, before system execution. They allow to determine which parameters’ values are suited to a particular situation. The authors illustrate the approach using a case study of dynamic websites performance. The goal is to maintain two metrics, the CPU use and the memory use, below certain thresholds. The adaptable parameters are the timeout and the keepalive of the configuration of Apache web server. By adapting these parameters, it is possible to increase or decrease the use of CPU and memory. The system monitors continuously both metrics and they are input to the equations, which output the parameters’ values.

In the aforementioned approaches, the modeling of adaptive behavior is conducted at a relatively low-level of abstraction. In particular, when the aim is to use self-adaptation in the context of autonomic computing, as pointed out in [Kep05], it is essential that humans can express their system’s goals in an easy way, at an adequate level of abstraction. With this objective in mind, policies are considered a better choice than the models referred previously [Slo94].

The advantages of policy-based adaptations result from their declarative nature (in contrast with operational nature of the other referred models). This facilitates the understanding of the adaptation logic of the system by an operator, without requiring full detailed knowledge of implementations. The use of policies allows to achieve independence from the current system state, facilitating and reducing development effort and subsequent tuning of the system’s adaptive behavior. Next, are addressed the two main types of policies used for expressing adaptation: action-based and goal-based.

2.2.2.1 Action Policies

Action policies [KW04] are declarative situation-action rules that express the actions that should be performed when given conditions are satisfied. In the context of self-adaptive systems, they are used to govern the adaptation of the managed system. Each rule indicates a set of actions that should be taken in response to events (or in particular states) that indicate the need of adaptation. Several policy languages for defining action-policies have been proposed. The structure of these policies ranges from a simple if-then-else format [KC03a] to a more
sophisticated event-condition-action (ECA) [MD89].

From the existing policy languages, Ponder [DDLS01] has achieved a renowned position by its completeness and comprehensive support. The language, primarily offering different formats to describe access control rules, has been extensively used for adaptation purposes [MLS04, FLR10]. The language allows to specify different behaviors for each system object. A rule determines how to choose the best behavior for the current scenario. Each rule is triggered by a specific event and includes a condition clause that must be respected and the respective action that must be performed. Events signal new circumstances, changes, that can be either internal to the system or external, which must be addressed. For example, in a website, if a login attempt fails the user/password tuple three times, an event is generated. Conditions are predicates that must be evaluated to determine if the rule applies or not, and which action will be chosen. The actions describe the adaptations that will change the behavior. Other policy languages exist that target specific adaptation goals, for example, targeting network management [SK05] or dynamic provisioning of resources [ZJY+09].

PBAAM, already mentioned, is an example of an approach to self-adaptation that uses action policies to specify how the structure changes. They consist of a set of observations and a list of responses; when the list of observations is fully satisfied, the entire set of responses is enacted. To facilitate the understanding, they may contain a human readable textual description that indicates the purpose of the policy to operators.

Action-based policies present some drawbacks that can be hard to tackle. One drawback is that policies require that the adaptive behavior can be specified and analyzed before system deployment. Another drawback may be the amount of knowledge that an operator has to gather before being able to describe the system adaptive behavior in terms of situation-action rules. This requires policy makers to be intimately familiar with low-level details of system function. Also, the larger the number of adaptations, the more complex and error-prone becomes the task of specifying the policy. Another drawback is that, if there are many elements of the context that can change independently, the number of rules necessary to describe the adaptive behavior can quickly become very large and complex to be manipulated by humans. Namely, it may become impossible for a human to predict the combined effect of potential conflicting rules. Finally, if the adaptive behavior changes or further context information is added, the policy may have to be written from scratch, with few reuse options.

2.2.2.2 Goal Policies

To address the drawbacks of action policies, some policy-based approaches employ high-level or abstract goals. The idea is to support the definition of how the system should behave without detailed low-level understanding of what the system does. The approach is responsible to determine the actions required to achieve the goals.

Goal policies can be specified in numerous ways. In the three-layer reference model [SHMK08], a goal policy is expressed as a set of formulas, using temporal logic. These
goals, together with a description of the system capabilities, are used to generate action policies to enforce the goals. This generation relies in identifying all the states from which it is possible to lead the system to a correct state, thus creating a rule for each undesired but amendable state. Other approaches maintain action policies and rely in a mechanism to map goal policies to action policies [BLMR04]. A number of approaches rely in adaptation impact estimation to generate action policies [BB08]. Reinforcement learning and heavy computations allow to estimate the results of executing an adaptation in a certain state, thus selecting or not an adaptation to address a state. Reinforcement learning allows the adaptation logic to gather feedback on the impacts of performing an adaptation, and use that same information in future estimations.

Another approach to the specification of goal policies is to use utility functions [TK04, FHS+06, CGS06]. The idea in this case is to define the utility (a scalar value) of each possible system configuration as a function of specific data available in the context (e.g., memory and bandwidth available). The aim is to assemble a configuration tailored to the current situation. For instance, in MADAM, a goal policy is expressed in terms of an utility function that assigns a scalar value to each possible system variant, as a function of the system properties in a given context. When the system needs to be adapted, the choice of a system variant relies on property predictor functions over the associated system properties.

2.2.3 Adaptation Support

The adaptation support assists the adaptation process, by providing necessary input and by carrying out the output of the decision making. The input is a number of informations necessary to assess, according to the model of the adaptive behavior, which adaptation is necessary, if any. The output is the selected adaptation to correct the system behavior. The input provisioning overlaps with a monitoring and analysis task and the output implementation with the execution task.

The implementation of the adaptation support depends on many system aspects, namely, the components, the system model, the adaptations, and the monitored context information. In the next sections, two frameworks that target adaptive systems are covered. The adaptation support provided by both frameworks is analyzed and their main concerns while doing it. The cactus framework was chosen due to its similarity with one of the case studies, while the Rainbow framework was chosen by its use of architectural system models.

2.2.3.1 The Cactus Framework

The Cactus framework [HSUW00] is a toolkit for the development of services and network protocols that can be adapted during runtime. This framework focus on fault-tolerance and survivability\(^3\).

The framework does not have an explicit system model. The system’s building blocks are

\(^3\)A system’s ability to continue regular execution even in face of attacks or failures.
adaptable and distributed services and protocols. These building blocks are compositions of software modules, each providing a different property or function. The composition can be adapted by changing the execution parameters of a software module or exchange the software modules altogether, thus changing the service composition.

In terms of adaptation support, Cactus offers monitoring and analysis support, and adaptation execution. This support requires that the building blocks are transformed in adaptive services. An adaptive service is a composition of adaptive components (ACs) (and occasional non adaptive components). An AC, depicted in Figure 2.3, is a collection of software modules and a component adaptor. The latter coordinates adaptations between the software modules. Each software module provides a different implementation of the component functionality, with the component adaptor switching between the alternative modules or changing the execution parameters of the component.

![Adaptive Component in Cactus](image)

In terms of monitoring, Cactus allows the construction of monitors, as a collection of micro-protocols, to capture relevant context information, such as resource consumption, and report that information. The information can be handed over to component adaptors or to the user through a GUI.

In terms of execution, Cactus has to address several issues related with the state and the distribution. When switching software modules, it may be necessary to transfer state from the old module to the new one. Therefore, using service variables, Cactus allows to transfer state without trouble, since the service variables are available to any software module in the composition. Furthermore, if the service is distributed, adapting it may require further care. Cactus relies in a three phase process to ensure the coordination of the different sites [CHS01]. The process begins with detection where the component adaptor determines if the current module is the best, and, if not, which one is. This is done using fitness functions that take the current system state as input. The next phase is the agreement between sites to adapt, where all hosts reach a conclusion if it is necessary to adapt and how. This is achieved using consensus. Finally, the last

---

4 The components and services are adaptive because the component adaptor includes the adaptation logic for that component.
phase is action, where a graceful adaptation from one module to another is performed. These concerns may not apply when changing the service parameters.

Overall, the Cactus framework provides a rich adaptation support for distributed services and network protocols change. The system model represents the system in terms of its components and allows to change parameters and exchange modules. The framework employs a control loop where a number of monitors control the context information and component adaptors perform adaptations, taking care of state transfer and coordination. However, there is not a complete separation of concerns since the adaptation logic is together with the adaptation execution, in the component adaptors. This may raise a number of difficulties when it is necessary to change the adaptive behavior.

### 2.2.3.2 The Rainbow Framework

The Rainbow framework [GCH+04] provides support to the self-adaptation of software systems. This framework focuses on providing a reusable infrastructure and low maintenance and development costs.

The framework uses an architectural system model, represented as a graph, where the components are nodes, the interactions between components are arcs, and also their properties of interest. The framework is not specific regarding which adaptations are supported. They are of the entire responsibility of the system developer.

In terms of adaptation support, Rainbow implements a closed-loop control system. This control loop is external to the system, thus, it can be reused by different software systems. This degree of separation is far greater in comparison with the Cactus framework.

![Adaptation infrastructure in Rainbow](image)

Figure 2.4: Adaptation infrastructure in Rainbow

The adaptation infrastructure is divided in three different elements, as depicted in Figure 2.4. The system layer establishes the connection between the infrastructure and the system, in other
words, it is the system access interface. This layer encompasses sensors (probes) to monitor the system and effectors to execute adaptations to the system. The architecture layer is responsible for the analysis and planning tasks. This layer has a model manager that gathers information from all sensors and interprets it and updates the system model representation. Furthermore, a constraint evaluator checks the system state to detect anomalies. When anomalies are detected the adaptation engine determines the adaptation action and conducts its implementation. Finally, the translation infrastructure mediates the mapping of information across the abstraction gap between the system to the model. The infrastructure also maintains a repository of the mappings from architectural-level elements to system-level elements and vice-versa.

The adaptation infrastructure depends from system-specific adaptation knowledge, which is hardly reusable from system to system. The adaptation knowledge is in fact the adaptation logic, which includes the system operational model. The operational model imposes explicit constraints to the system behavior, defining, among other elements, adaptation strategies. Adaptation strategies define which actions are possible and how they should be implemented for particular system concerns. Rainbow relies in architectural styles to improve the reusability of this knowledge. Architectural styles can be used by different systems that share the same structural and semantic properties. Allied with adaptation strategies, it is possible to define an adaptation style which allows to adapt any similar systems that share the same system concerns.

Overall, the Rainbow framework provides adaptation support that can be reused by different systems to achieve self-adaptation. The entire support is external to the system, requiring that the system provides an access interface. There is a clear separation between the adaptation logic and the adaptation support. The description of adaptation actions also includes its implementation, since the framework does not support generic adaptations.

### 2.2.4 Summary

The previous sections made a digest of the main concerns in choosing a system model, in adopting an adaptive behavior model, and in supporting adaptation, while assuming separation of concerns and a control layer. In terms of system model, the choice is determined by the type of adaptations and which system information will trigger them. To adapt it is necessary to have information on the current state of the adaptation targets. In terms of adopting a model of adaptive behavior, the concern is to allow a human-friendly specification of the adaptation logic. Policies present a number of benefits for the thesis problem. The support of adaptation brings a number of concerns, in component-based systems, mainly related to state transfer and coordination. The proposed approach addresses these concerns while proposing new techniques for the self-adaptation of composed systems.
Chapter 3

Proposed Approach

This chapter describes the proposed approach. First, the architecture is addressed, which follows the best practice design principles described in the previous chapter. Next, the case studies that will be used to exercise the proposed approach are presented. The work done so far is detailed in the next chapter.

3.1 Reference Architecture

The proposed approach follows the separation of concerns paradigm and relies in a feedback control system to manage the system, as previously addressed in Sections 2.1.2.1 and 2.1.2.2, respectively. The approach targets systems that are built from several adaptable components. These components can be services, protocols, or middleware, among others. It is assumed that adaptable components provide access interfaces that allow to gather context information or change their behavior (for instance, setting configuration parameters with new values). Furthermore, when structural adaptation is considered, it is assumed that the system supports this type of adaptation in during runtime. For instance, Cactus provides such functionality.

The proposed approach already reflects lessons learned from the case studies under development. Among these, the most important concerns the use of policies. In the beginning, when first designing the approach, one of the goals was to automatically compose an action policy from pre-existent collections of ECA rules. This selection of rules would be done according to the system structure and some behavior goals. However, our experience has shown that it is very hard to provide collections of rules that anticipate all the scenarios and behaviors before runtime. Furthermore, this approach requires a global view of the system to establish behavior goals and a detailed understanding of the adaptable components, their impacts, adaptations and impacts in the overall system behavior. It is extremely hard for a human operator to manage this complexity in a system composed of adaptable components.

As a result of these observations, the proposed approach considers two types of policies: action and goal policies. This solution addresses the drawbacks of action policies and achieves
a new level of abstraction, beneficial for complex systems. The action policies are used when i) the number of components is small enough that the operator is able to describe an effective policy; ii) it is possible to analyze and identify all the scenarios and adequate adaptations for each component, before runtime; and iii) legacy action policies already exist. The goal policies are used when i) the operator does not have detailed knowledge of the system components and their adaptations, but has a global view of the system and its behavior; ii) there is a large number of components and becomes very hard to manage the adaptive behavior; and iii) the system structure changes frequently and significantly during runtime. Both types of policies will be addressed in more detail in Sections 3.1.2.4 and 3.1.2.3.

To support both types of policies, the approach relies on both offline support and online support. The offline support handles goal policies. It is able to generate an action policy from the goal policy, specifications of possible adaptations, and their impact in the behavior. The online support handles solely action policies, which are evaluated when a change of behavior is necessary. The context monitoring and the execution of adaptations is also performed by the online support. Both offline and online support are assisted by a number of models that provide specific information and knowledge about adaptation, context, and mappings between different abstraction levels for both. Next, the models, the offline support, and the online support are described.

### 3.1.1 Models

The approach relies on a number of models that capture different key aspects related with the adaptive behavior and the adaptation support. The models describe information relevant to support self-adaptation, that depends of the system characteristics and execution environment. There are several models: component, structural, application, context, sensor, and effector models. This collection of models is called adaptation model and it is used by the change management and action policy layers.

The component model describes the relevant properties of each individual component, including behavioral adaptations. These properties include: i) context information provided by the component that may be sensed; ii) key performance indicators that apply to the component; iii) description of adaptations that can be applied to the component, and performance estimators and impact functions that estimate the impact of a change on the component’s performance indicators; and iv) constraints that must be satisfied when using the component, such as, other components and their configurations.

The structural model describes structural adaptations for the system and their impacts in the system’s key performance indicators. This encompasses adaptations that change components interaction and the system composition (for instance, to update a component with a newer version). Furthermore, it is also relevant to define the scope of a structural adaptation. For instance, if a component is used to provide different services to the application and/or used in different nodes of a distributed application, the scope defines which services/nodes are affected.
by the adaptation.

The application model describes the relevant application properties. These properties include: i) context information provided by the application that can be captured, such as user preferences; ii) the application’s key performance indicators; and iii) the components or compositions of components used by application.

The context model describes the context information that is relevant for the system adaptation. The model consists in a set of observables and events with context information. These are provided by the context analyzer to the policy interpreter, and can be used in the definition of the action policy. Typically, this information is obtained by interpreting, combining, or constraining data obtained from different sources. The ultimate context information is produced from the sensed data through a number of computations.

The sensor model describes the information provided by individual sensors. Sensors may gather information from the environment (for instance, from the operating system) or from individual components (for instance, a communication protocol may provide information about the measured packet loss), or the application. This model consists of a set of observables and events, that model the sensed data that needs to be provided to the context analyzer by appropriated sensors.

The effector model describes the commands that can be performed on each individual component to change its behavior (for instance, to change a parameter or an operational mode) and on the composition framework (for instance, to activate, deactivate components, or change the way components are connected). The effector model allows to transform an adaptation action and the selected adaptation strategy into a number of change directives.

3.1.2 The Offline Support

The offline support uses a number of models that characterize the system and a goal policy as an input, and generates an action policy, as illustrated in Figure 3.1. The next sections describe the goal policy, the policy generator, and the resulting open action policy.

![Figure 3.1: The offline support](image-url)
3.1.2.1 Goal Policy

The goal policy is an ordered collection of goals, that establishes bounds and target values for key performance indicators (KPIs). Six different types of goals were identified, divided in two categories: exact and approximation goals. Exact goals separate the values of a KPI in two disjoint sets: acceptable and not acceptable. The following types of exact goals are considered:

- **Goal** goal_name: kpi_name Above threshold_down MinimumGain gvalue
- **Goal** goal_name: kpi_name Below threshold_up MinimumGain gvalue
- **Goal** goal_name: kpi_name Between thr_down thr_up MinimumGain gvalue

An *Above* goal states that the value of the KPI should be kept above the stated threshold, a *Below* goal that the value should be kept below the threshold, and a *Between* goal that the value should be kept within lower and upper thresholds. In all three, the *MinimumGain* specifies the minimum change necessary to perform the adaptation; that is, if the estimated change in the KPI value is below *gvalue*, the adaptation is not worth performing. The *gvalue* should be greater than the error margin specified for the target KPI.

In contrast, instead of simply classifying the values of a KPI as good or bad, approximation goals specify a total order between these values, that is, for any two values, it specifies which one is better. The following types of approximation goals are considered:

- **Goal** goal_name: kpi_name Close target MinimumGain gvalue Every interval
- **Goal** goal_name: Minimize kpi_name MinimumGain gvalue Every interval
- **Goal** goal_name: Maximize kpi_name MinimumGain gvalue Every interval

A *Close* goal states that the KPI value should be kept as close as possible to the target value, a *Minimize* goal states that the KPI value should be as small as possible, and a *Maximize* goal states that it should be as large as possible. As with exact goals, it is also possible to specify the expected minimum gain required in order to perform an adaptation. Furthermore, associated with each approximation goal, is a time *interval* that specifies how often the system should try to find an adaptation aiming for a better value for the KPI. Note that while adaptation towards an exact goal is only triggered when the current KPI value is unacceptable, an approximation goal opens the possibility of continuously attempting to improve the system behavior aiming for a better value.

3.1.2.2 Policy Generator

To produce an open action policy, the generator uses the goal policy and information provided by the models. This process executes in two steps. The first step is the event extraction, where the generator extracts, for each goal, one or two events at most, depending on the goals. The events characterize scenarios where an adaptation is required to satisfy the goals. These events are supported by the context model, that describes how to compute them. The mapping between goals and events is shown in Table 3.1.
The second step is the selection of adaptations whose impact is *favorable* to a scenario. A favorable adaptation is one whose impact brings the KPI closer to the associated goal. This information depends on the adaptation’s impact functions. However, since no state information is available, the concrete estimated impact is not known. Therefore, although it is possible to know if the adaptation will bring the KPI closer to the goal, it is impossible to ascertain if it will be close enough to correct the behavior and satisfy the goal. The result is that for an event there are several possible adaptations, whose final selection will depend on the current state.

<table>
<thead>
<tr>
<th>Type</th>
<th>Goal</th>
<th>Event 1</th>
<th>Event 2</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>Above</td>
<td>kpiBelow(kpi, x)</td>
<td>-</td>
<td>threshold exceeded</td>
</tr>
<tr>
<td>Exact</td>
<td>Below</td>
<td>kpiAbove(kpi, y)</td>
<td>-</td>
<td>threshold exceeded</td>
</tr>
<tr>
<td>Exact</td>
<td>Between</td>
<td>kpiBelow(kpi, x)</td>
<td>kpiAbove(kpi, y)</td>
<td>threshold exceeded</td>
</tr>
<tr>
<td>Approx</td>
<td>Close</td>
<td>kpiIncrease(kpi, θ, cond)</td>
<td>kpiDecrease(kpi, θ, cond)</td>
<td>periodic</td>
</tr>
<tr>
<td>Approx</td>
<td>Maximize</td>
<td>kpiIncrease(kpi, θ, cond)</td>
<td>-</td>
<td>periodic</td>
</tr>
<tr>
<td>Approx</td>
<td>Minimize</td>
<td>kpiIncrease(kpi, θ, cond)</td>
<td>-</td>
<td>periodic</td>
</tr>
</tbody>
</table>

Table 3.1: Events extracted for each type of goal

### 3.1.2.3 Open Action Policy

Unfortunately, it is not possible to select the single best adaptation(s) offline. Therefore, the generated policy is a *multi-option* open action policy. Open action policies are composed by rules that follow a structure richer than regular action policies, now referred to as closed action policies, by opposition. The rule is triggered by an event, in the same manner that regular ECA rules are. However, the rules do not include a separate *conditions* clause. The conditions are still attached to the adaptations. The list of adaptation actions is of all possible actions, including conflicts and dependencies between actions. Finally, the rules have extra information, by carrying a digest of the goals described in the goal policy in an *Optimizing* clause. The criteria are in the same order as the goals. This order is important for the policy interpreter, as will be made clear in the next section.

When event
- **Conflicts**: (S1.A, S2.X) **Dependencies**: (S2.Y, S3.Z)
- **Optimizing** criterion_1, criterion_2, ...

### 3.1.2.4 Closed Action Policies

The closed action policies are manually specified by a human before runtime. Therefore, despite not being automatically generated like the open action policies, they are also produced offline. Closed action policies have the typical ECA structure. The rules have an event, that triggers the rule; some conditions, that have to be validated; and adaptation actions, that will be applied (all of them) if the conditions are valid. Furthermore, a closed action policy can have more than one rule triggered by the same event (but with different conditions), while open
policies, only have one rule per event. In distributed settings, the adaptation can also have a scope, which lists the targets of the adaptations, such as nodes or compositions. A typical rule of a closed action policy is:

\[
\text{When event } \quad \text{Conditions: condition}_1, \text{ condition}_2, \ldots \\
\text{Actions: S1.A, S2.Y, S3.Z} \\
\text{Scope: node1}
\]

3.1.3 The Online Support

The online support self-manages the system, by monitoring, evaluating, and adapting. The online support has a representation of the system in a system model, that is updated when the system suffers any behavioral or structural adaptations. The online support is separated in three layers of abstraction: the managed system, the change management, and the action policy layers. The first two layers are similar to the three-level robotic architectures, since they do not consider the managed system. The same two layer make use of the adaptation model. This architecture is depicted in Figure 3.2, and employs the following notation:

- Each layer is depicted by a rectangular frame.
- Boxes inside the frame represent components that execute at that layer.
- An upwards arrow leaving the frame represents information that is provided in runtime by the components of the layer to the layer above.
- An downwards arrow entering the frame represents directives received from the layer above.
- “Stickers” in the side vertical bar represent models that are used in different layers of the architecture.
- In components, the $S$ stands for sensors and $E$ stands for effectors.

3.1.3.1 Managed System Layer

The bottom layer is the managed system. As the name implies, this corresponds to the actual system that needs to be adapted. It includes the software components and their execution environment, such as operating system, middleware, etc. The components, the operating system, and the middleware provide low-level sensed information to the layer above, such as actual resource consumption (cpu utilization, free memory, etc), network conditions (latency, packet drop rates), other environmental conditions (temperature, locations, etc), as well as any other information that may be useful to characterize scenarios where adaptation is necessary. Also, the layer provides the low-level directives that allow to change the behavior of the components,
3.1. REFERENCE ARCHITECTURE

Figure 3.2: Three levels of abstraction of online support

the operating system, or the middleware (for instance, to increase the priority of processes, to change the operational mode of a component, etc).

3.1.3.2 Change Management Layer

The middle layer is the change management which monitors the context information and controls adaptation execution. The context analyzer handles the context, transforming low-level sensed data in high-level context information. The context analyzer also identifies relevant changes, which are passed to the top layer. The adaptation planner controls the adaptation, receives the selected adaptation, and determines the best way of execute it. First, by selecting the most appropriate adaptation strategy and, then, by translating the pair adaptation–strategy to a sequence of change directives. Overall, the layer has three main roles:

- It analyses and interprets the sensed data to identify relevant changes in the context;
• It maintains the system model, a representation of the actual system configuration. This representation includes the components, their interactions, and the configurable parameters, thus, allowing structural and behavioral adaptations;

• It defines how a given adaptation should be applied to system. In order to perform an adaptation it is usually necessary to execute a sequence of commands on effectors, possibly in different nodes if the system is distributed, which is referred to as an adaptation strategy. The selection of the most appropriate strategy for each adaptation is automated using component and structural models and a library of available strategies built in the adaptation planner.

3.1.3.3 Action Policy Layer

The top layer action policy is the decision making mechanism that controls the system behavior. This layer consists of a policy interpreter that accepts an action policy. As described in Section 3.1.2.3, the action policies that result from the generation process are slightly different from regular ECA rules. Therefore, the policy interpreter accepts both closed action policies and the open action policies.

The policy interpreter handles the two policies in different manners. If it is a closed action policy, the interpreter starts with the first rule that is triggered by the relevant changes and validates the conditions. If the conditions are valid, then, the adaptation actions of the rule are fed to the middle layer. Otherwise, the interpreter searches for another rule or, if there is none, no adaptation is performed.

On the other hand, if it is an open action policy, extra computation is necessary. The policy interpreter begins by estimating the impact of adaptations in the triggered rule. The estimation is based on the system current state and on the impact functions of each adaptation. Next, the interpreter tries to optimize the criteria specified in the rule. The order matters here. The criterion that comes first has priority. Only after satisfying one criterion, the algorithm will move to the next one, otherwise, it will stop at that criterion.

3.2 Case Studies

As noted in Chapter 1, to evaluate the proposed approach in different settings, the thesis will consider two different types of systems, namely:

• Distributed systems composed of fine grain components.

• Centralized systems composed of coarse grain components.
3.2. CASE STUDIES

3.2.1 Case study 1: Micro Protocol Stacks

Micro protocol stacks will be used as a case study for applying our approach to distributed systems composed of fine grain components. In this system, each individual component is a micro protocol implementation that executes in one of the multiple nodes of a distributed system. There are several frameworks that support protocol stacks, such as Cactus. This work will target the adaptation of protocols and compositions of protocols to address concerns related to load, latency, and throughput improvement at the expense of quality of service.

3.2.1.1 Managed System Layer

The group communication protocols will be used as a case study, given that they support the communication among multiple processes. This creates more challenging scenarios for runtime adaptation, as often is required to coordinate various nodes. Micro protocols can be composed in protocol stacks, where the protocol of layer \( n \) provides services to the protocol of layer \( n + 1 \) using the services of protocol \( n - 1 \). The term “micro”, stems from the fact that a stack implement using this approach can easily include as many as 20 micro protocols. An instance of a protocol stack in called a channel. Typically, an application uses multiple channels with different properties (for instance a data channel and a control channel). There are multiple protocol composition and execution frameworks that support this type of composition. The case study is based on the Appia framework [MPR01].

In this setting, the context is provided by sensors executing in each node, that collect information such as traffic load, network latencies, message losses, etc. Effectors can locally change parameters in running micro protocols or change, on the fly, the protocols stack that implement the channels used by the application (by adding, removing, or replacing protocols). This is achieved using the reconfiguration features of a version of the Appia system — the RAppia — that is being developed in the context of this thesis.

Another challenging aspect of the micro protocol case study is the definition of performance estimators and impact functions for communications protocols, an area that has been, to the best of our knowledge, unexplored in the literature.

Relevant context information to be used in the definition of an action policy for micro protocol compositions are values such as the number of active participants, traffic load imposed on the system, round-trip times, packet drops ratios, observed latency in the communication, observed throughput, etc. Significant changes in these values may require the adaptation of one or more micro protocols in one or multiple nodes in the system.

3.2.1.2 Change Management Layer

For this case study, there are two main challenges in the development of the change management layer. One is to define the scope of the adaptations (as multiple instances of a component
may execute, not only in different nodes but also in different compositions). The second challenge is to select the most appropriate strategy to perform an adaptation. A widely used strategy to adapt communication stacks is to place the entire stack in a quiescent state, and perform the adaptation at all nodes when the stack is idle, in a coordinated manner. Unfortunately, this strategy enforces a temporary disruption of the communication flow. Therefore, it is interesting to use more efficient strategies, that require less coordination, whenever is possible.

3.2.1.3 Action Policy Layer

In this case study, the throughput and the latency are indicators of the system performance and state. Increasing latency and decreasing throughput are signs of increasing load in the system or overload. To address the increasing load, one possibility is to decrease the number of messages passed in the system. This can be achieved by changing algorithm or aggregate messages together. An action policy would have to identify which scenarios of latency and throughput denote overload, and assess which algorithm is best for the current scenario. For instance, in group communication, a total order protocol is employed. There are various implementations of total order algorithms, some perform better in specific scenarios, when compared to others. For instance, the sequencer-based total order protocol [DSU04] is one that performs best when only one member is sending messages. The next rule illustrates an adaptation that would take advantage of a token total order protocol, that fares better when there is congestion in the network.

\[
\text{When NetworkCongestion} \\
\quad \text{With } !hasservice \text{ (TokenTOService, DataChannel)} \Rightarrow \\
\quad \text{Do exchangeService ([TokenTOService])} \\
\quad \text{For TotalOrderService}
\]

The rules describes an exchange of total order protocol, from the current to a token total order protocol, that only deploys it if it is not already in use.

3.2.1.4 Goal Policies and Generator

In the case study of micro-protocols, it is also possible to describe the desirable behavior of a communication channel using high-level goals. An example of such goal policy is:

\[
\text{Goal limit latency: latency Below 2 MinimumGain 0.05} \\
\text{Goal max throughput: Maximize throughput MinimumGain 10 Every 600}
\]

The goals state that the latency should be maintained below a given threshold and that the throughput should be maximize whenever it is possible. However, in order to generate an action policy that reflects such high-level goals, adequate impact functions for the adaptations are necessary.
3.2. CASE STUDIES

3.2.2 Case Study 2: Dynamic Websites

The dynamic websites case study will be used as a case study for applying our approach to centralized systems composed of coarse grain components. Dynamic websites are websites where the webpages are dynamic. They can be periodically generated or generated for a particular request. These websites can rely in several services to generate the webpage returned to the client browser. Examples of dynamic websites are e-commerce websites, such as Amazon.com, or review and rating websites, such as TripAdvisor.com. The case study will address adaptation management in the server side or backend.

3.2.2.1 Managed System Layer

In websites, services are provide to client using a web based interface, typically using the HTTP protocol. The services are implemented by programs that capture the business logic, instantiated as CGIs, servlets, ASPs, or similar technologies (from now on, these programs will be referred to as scripts). In turn, scripts may invoke some services from the back end (most often, but not necessarily limited to, a database and a file server). Quite often, a given service may be provided with different trade-offs between amount and accuracy of the information provided (denoted the harvest) and the resource consumption required to provide such information. For instance, a site can return higher quality images at the cost of additional network and CPU consumption. This creates opportunities for dynamic adaptation, to respond to changes in the demand. For instance, consider a service that provides web pages that include several images. When the demand is very high, and server resources become under stress, one can replace a script that provides high quality images by a script that provides a low resolution version of the same images. Example of relevant context information in dynamic websites are resource consumption values, such as CPU and memory utilization. Impact functions should provide an estimate of the effect of the adaptations on these parameters.

3.2.2.2 Change Management Layer

Many adaptations can be implemented by replacing one script by another script. Also, it is often possible to start serving new requests using a new script, even if requests that started to be served before the effector command has been applied are still running. Thus, the experimentation of sophisticated strategies is not expected in this case study. On the other hand, the impact functions of adaptations, in particular when components are not independent, may be very challenging.

3.2.2.3 Action Policy Layer

In dynamic content websites, key performance indicators include the harvest of each request served, the system throughput, and the end-to-end delay. Since resources are limited, when the
load increases one may be faced with the need to reduce the harvest to maintain satisfactory levels of throughput and delay. An action policy for dynamic websites has to state for which services the harvest may be sacrificed first and the minimum throughput values and maximum delay values acceptable for each service. The selection of adaptations has to consider the current resource consumption and load for each service. An adaptation to the service that is processing the largest number of requests may have a much higher impact on the system global behavior, than an adaptation on a service that is serving few requests.

Describing closed action policies for dynamic websites may become very hard. To manually capture all the trade-offs involved in resource provisioning, fluctuating demands, peak loads, among other issues becomes increasingly hard, as the number of services increases. This approach was attempted in [FLR10].

3.2.2.4 Goal Policies and Generator

The goal policies will include goals that try to maximize the quality of service and maintain the resource consumption below certain threshold, to allow enough buffer to survive to peak loads and rapidly growing load or overload. For instance, the following rules describes the goals of trying to maximize the image resolution while keeping the CPU use below a threshold.

\[
\text{Goal } \text{limit_cpu: cpu } \text{Below} \ 0.6 \ \text{MinimumGain} \ 0.05 \\
\text{Goal } \text{max_resolution: Maximize resolution MinimumGain} \ 1 \ \text{Every} \ 300
\]

The open action policies are generated using the method described in the reference model. For example, to keep the CPU use below 0.6, the result would be: when CPU use is above 0.6 do favoral actions.
Chapter 4

Work Progress

This chapter summarizes the work progress until April 2010. The approach described in the previous chapter is the result of the refinement of ideas that have been incrementally explored using the two case studies. As noted in the introduction, initial work addressed the problem of adapting compositions of micro protocols. This work was later extended to address the adaptation of coarser grain components with the dynamic websites case study. Most of the work has been captured in published papers, namely [RRL07a, RRL07b, RLR08, RRL+09], ordered chronologically. In the following paragraphs, the problems addressed by each of these papers are briefly described.

4.1 Paper [RRL07a]

The research was initiated using the Appia protocol composition framework. Appia supported the configuration of protocol stacks at deployment time but not during runtime. Therefore, the first step was to study the challenges involved in augmenting the system with support for dynamic adaptation. This work was particularly useful to get deeper insights on the problem at hand. The results were published in the paper Building Adaptive Systems with Service Composition Frameworks [RRL07a]. The paper did the first steps in the definition of an adaptation model for the Appia framework and identified a number of limitations of the protocol composition framework that were an impairment to the efficient support of dynamic adaptation. It laid the basis for the development of a version of the Appia framework called RAppia. In detail, the paper includes the following contributions:

- The system model is addressed in part in this paper, namely, the concern of organizing the protocols in hierarchies of types, to represent them.

- Context sensors are described. The paper describes a generic context sensor, namely how it captures data from micro protocols and how it sends the data to the context analyzer.

- Component effectors are also described. The paper details how an effector is able to adapt
a stack of micro protocols and how it handles quiescence, state transfer, and is controlled by the adaptation planner.

- The RA ppia fra mework is also partly described in this paper. The paper outlines the main obstacles to runtime adaptation in the Appia framework and describes the chief modifications made to the original framework.

### 4.2 Paper [RRL07b]

The paper A framework to support multiple reconfiguration strategies [RRL07b] is the result of the work in developing a prototype of an the RA ppia system and of researching different manners to make dynamic adaptation more efficient. The paper describes a library of adaptation strategies to execute different adaptation actions in distributed settings. Furthermore, the paper also addresses the selection of the best strategy for an action, in the context of micro protocol stacks. More precisely, the following elements are described:

- The system model is also implicit in this paper, namely, the structure of the system and where each micro protocol is located.

- The library of adaptation strategies contains a number of adaptation strategies that are pairs of orchestrations and local reconfiguration techniques.

- The context analyzer is addressed by the first time in this paper. The paper describes its role and responsibilities in the online support.

- The adaptation analyzer is addressed by the first time in this paper. The paper describes its role in performing the adaptation strategy.

- The effector model is briefly addressed, namely, the change directives used by each strategy are described.

- The policy interpreter is briefly introduced in the paper, describing the closed action policy validation.

- A closed action policy is specified for the example presented in the paper.

### 4.3 Paper [RLR08]

The paper Modeling adaptive services for distributed systems [RLR08] is the result of the experience gained from the previous papers and refines the adaptation model. The paper describes the entire collection of models, with the exception of the effector and structural models, for the dynamic adaptation of compositions of micro protocols.
4.4. **PAPER [RRL⁺09]**

- The *system model* is again addressed in this paper, giving more details about each component representation.

- The *component model* is described and a specification syntax is proposed, that addresses the configurable parameters and which context information can be captured.

- The *context model* describes the exported observables and events, and how they are obtained.

- The *sensor model* is described and a specification syntax is proposed, that addresses the captured observables and events.

- The *application model* describes which stacks of protocols the application has, using a specific syntax that also includes any relevant context information produced by the application.

#### 4.4 Paper [RRL⁺09]

After working on the dynamic adaptation of micro protocols, the lessons learned were applied to the dynamic adaptation of dynamic websites, in an attempt to improve it. The observation that closed action policies where very hard to derive manually, lead us to work on an offline support to automatically derive action policies from goal policies. This work is described in the paper *From Local Impact Functions to Global Adaptation of Service Compositions* [RRL⁺09]. The paper describes the offline support elements and process, together with the selection algorithm used by the policy interpreter for open action policies. Although the emphasis of the paper is on the offline support, an instantiation of the architecture for the dynamic websites case study, including the development of sensors, effectors, and their respective models, had to be developed to perform an experimental evaluation. This work is also briefly reported in the paper. In detail:

- The *goal policy* is presented for a website that relies in three services. The types of goals and the specification syntax are presented.

- The *policy generator* is described in this paper and uses the goal policy.

- The *open action policy* is the result of the policy generator and is also described in this paper. Its syntax is addressed.

- The *policy interpreter* is described in this paper, namely, the adaptation selection algorithm for open action policies.

- The *structural model* is described and a specification syntax is proposed, that addresses the possible adaptations, their impacts and requirements.
• The models are assumed in this paper, although not explicitly addressed.
• The adaptation planner is assumed in the paper but without adaptation strategies.
• The context analyzer is assumed in the paper but not explicitly described.

4.5 Ongoing work

After the last publication, the online support for the dynamic websites case study was enhanced and the policy interpreter was refined. In the former, the accuracy of the monitoring system was improved, in terms of sensors and context analyzer. In the latter, the policy interpreter’s selection algorithm was further tested using a richer dynamic website example. The results of this work are not yet published but a draft of the technical report has been produced.

Currently, ongoing work is focused on the support of goal policies in the micro protocol case study. This work is going to be performed in cooperation with a master student, that is implementing and evaluating a more robust version of the RAppia prototype.

If time allows, it would also be interesting to address the issue of dependencies between services, when deriving open action policies for dynamic websites. Namely, to determine which modifications to the policy generator, policy interpreter, and adaptation planner are necessary, to take into account such dependencies.

The completion of the remaining work is expected in the end of 2010. The thesis deliver is intended for the Spring of 2011.
Chapter 5

Conclusions

This proposal proposes an approach to build self-adaptive systems composed by individual adaptable components, developed in a non-coordinated manner. The approach relies in two best practice design principles: separation of concerns and closed-loop control, and presents a number of techniques to facilitate the design and implementation. The approach adopts different models of adaptive behavior: action and goal policies, that together allow to overcome the limitations of each one separately. The approach also includes mechanisms to address distribution and handling components state.

The approach assessment relies in two different settings: micro protocol stacks and dynamic websites, which allow to evaluate and refine it. The work progress for each case study is detailed in the proposal, as well as a description of the ongoing work. The proposal concludes with the layout of the remaining work and expected conclusion date.
Bibliography


