A Reference Architecture for Component-Based Self-Adaptive Software Systems

Magister Graduation Project

Presented in partial fulfillment to obtain the Title of
Magister in Informatics and Telecommunications Management

by

Lorena Castañeda Bueno

Advisor: Gabriel Tamura

Department of Information and Communication Technologies
Faculty of Engineering

UNIVERSIDAD ICESI

2012

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, businesses based on software systems experiment diverse changes in their requirements of operation. Continuous changes in customer preferences and technology are affecting the business objectives, forcing the organizations to make decisions over the company's present and future. The organizations that have survived to this phenomenon are known as emergent organizations. However, along with these organizations, the software systems whose purpose was to support the organization's stability is now required to be flexible enough to support the changes and therefore, to be able to adapt in order to meet the new business goals [28].

For computing systems, the context is unpredictable, organizations are forced to change continually the business goals and therefore the software systems must meet those requirements in short times. Under these conditions, it is nearly impossible for software engineers to design an error free system. Therefore, engineers should also monitor and analyze the system every time searching for possible faults. Subsequently, most of the IT budget is spent in prevention and recovery from failures [6].

As a result, organizations need a cost benefit improved solution for the management of these complex systems and then release its administrators of this managing and routine tasks as far as possible. It is necessary to build systems that can be more self-manageable. In order to do that, the systems must have characteristics stated as follows [6]:

- The system needs to *know itself*, therefore, the components in the system must have an identity.
- The system must know the context surrounding its activity.
- The system must coexist with foreign components, that is, operating with open standards.
- The system should know its purpose and the business goals that it is pursuing.

As new business objectives change in organizations, software systems must be able to meet them by adapting some part of itself. For this adaptation to be performed by the system itself, it is required the execution of four tasks: **monitoring** the new business requirements, as the internal and the external context, **analyzing** the data gathered to identify symptoms and need for adaptation, **planning** a set of adaptive actions, and **executing** those actions over the system in execution [16, 6, 14, 3].

One of the most important challenges in self-adaptation is to create the ability in the system to reason about itself and its context. Control engineering uses feedback to measure and act upon the behavior of the system. Moreover, *feedback loops* become important in the process that the system must follow to monitor and analyze a desired property of the system and then evaluate if it is being reached [11]. Software engineers have made significant efforts to propose self-adaptive systems. These proposals have used different approaches based on feedback loops, components and even hybrid architectures, and communication standards [12, 15, 11, 16, 7, 20, 21, 30, 29, 27]. However, all these approaches are not sufficient to fit the different systems, data types, and configurations that are available to build self-adaptive software systems. Often, these proposals merge self-adaptation with the target system making very difficult to analyze the adaptation capability as an independent function, and feedback loops are not explicit in most of these proposals [15, 11, 16, 7, 20, 21]. In this scenario, a reference architecture for designing self-adaptive software based on feedback loops is a necessity.

The separation of concerns between the adaptation controller and the target system is important to be able to manage their own properties and analyze and assess them separately. Not observing this separation of concerns it is very difficult to identify and analyze the adaptation mechanism as an independent functionality. Moreover, the less visible feedback loops managing the adaptation in a system, the more difficult to implement the self-adaptation [18].

In this project we design a component-based architecture to implement the basic elements required for self-adaptation: *monitoring, analyzing, planning and executing*, independently from the system that is being adapted. Moreover, the components of this architecture can be distributed among different machines providing flexibility and extensibility. Our architecture will provide extensible capabilities in such a way it can be extended to be implemented as in e.g., the Service-Oriented paradigm. To evaluate our proposal, we implement the architecture and evaluate it in an specific case of study.

1.1 Project Goals and Objectives

1.1.1 General Objective

To design and implement a component-based reference architecture that defines the four basic tasks required for self-adaptation: Monitor, Analyzer, Planner and Executor including the Sensors and Effectors required to adapt component-based software systems.

1.1.2 Specific Objectives

1. To design a component-based reference architecture with distribution and extensible capabilities for self-adaptive systems according to the reference model for context-based self-adaptive systems [30]

2. To implement the proposed architecture as a Java software application.

3. To evaluate the proposal in a specific case of study using the service-oriented paradigm.

1.2 Document overview

The remainder chapters of this document are organized as follow: Chapter 2 describes the context and background of this project. This chapter also gives a brief overview of Autonomic Computing and Self-Adaptive Software Systems approaches as well as the methodology to derive a reference architecture, and the definition of a Component-Based Software. Chapter 3 presents the contribution of this project, the Reference Architecture for Component-Based Self-Adaptive Systems. This chapter describes also the architectural patterns applied, the data flow amongst the components and a description of each component intervening in each step in the adaptation process of a software system. Chapter 4 presents the evaluation of this proposal in an specific case of study using a story board proposed by Tamura *et al* [24] by using SoapUI. Finally, Chapter 5 presents the conclusions and a proposal for future work.

Chapter 2

Context and Background

As mentioned in Chapter 1, the separation of concerns is important to facilitate the management of adaptation as a software system independently the self-adaptive software system. To understand the relevance of this proposal it is important to define the ground basis for the main concepts involved in this project: Autonomic computing, Self-Adaptive Systems, Component-Based Software, Software Architecture, Reference Models and Reference Architectures.

2.1 Autonomic Computing

An autonomic system is a system that can manage itself given some kind of decision making mechanism, such as policies, from administrators [14]. An autonomic system is composed of many self-managed components that interact with each other autonomously. The relationship among these components must be controlled by themselves and with minimum human interaction. Furthermore, an autonomic system must be able to maintain and adjust their operations according to changing external or internal conditions, and guarantee the alignment with the business goals.

The fundamental characteristic of an autonomic system, is self-management, in other words, the capability to run routine tasks and maintenance without the intervention of the system administrator. The system must be continuously monitoring itself to be aware of changes in the system that might require either reconfiguration or optimization of the components, protecting itself against suspected wrong behavior or recovering from failures [14]. In order to accomplish this self-managing behavior, systems must implement four fundamental features: **self-configuring**, the ability to adapt to the shifting environment itself in accordance with high-level policies aligned to the business goals given by system administrators; **self-healing**, the ability to recover itself after a disruption in the system and minimize outages to keep the software system up and available; **self-protecting**, the capacity to predict, detect, recognize and protect itself against malicious attacks and unplanned cascading failures; and **self-optimizing**, as the way the system improves its operations [6, 14].

2.2 Software Adaptation

Adaptation refers to the capability of changing the software structure or behavior according to significant alterations in the environment. This adaptation must occur dynamically and at runtime. The adaptive behavior can be seen for instance on mobile devices, which must adapt to different connectivity conditions but keeping quality attributes such as availability and performance. Furthermore, this adaptive capability is necessary on the systems that are vital to survive to hardware failures and security attacks [16].

Self-Adaptive Software systems rely completely on the **context** which can be defined as any information that characterizes the state of entities that affect the execution, maintenance and evolution of systems[29].. Therefore, the desired software adaptation requires the system to be context-aware because in the context are the changes and behavioral information related to the decision and need of adaptation of the system, and because after adaptation occurs the context, as the system itself, has changed.

Software adaptation can be implemented either by parameter or by composition, among others. Parameter adaptation alters software model variables determining for the system behavior. However, it does not allow new components or algorithms to be added after deployment. On the other hand, compositional adaptation exchange algorithms or components with other elements in the system to meet the current environment conditions [16].

Dynamic adaptation provides flexibility to the system because it can adjust itself at runtime. However, this flexibility is very difficult to design and program for software engineers. McKinley *et al.* include three requirements to implement reconfigurable software: *separation of concerns, computational reflection and component-based design* [5, 16].

- **Separation of concerns** refers to the separation in development of an application's functional behavior from the non-functional requirements. It simplifies development and maintenance and promotes the reuse software components.
- **Computational reflection** refers to the software's ability to reason about its own structure and possibly to alter it. Reflection implies two capabilities: *introspection* that allows a system to observe its own structure, and *intercession* that allows the system to modify itself.
- **Component-based design** relates to the software units that are developed, deployed and composed by third parties independently. This technique supports two types of composition: *static*, where many components are combined to produce an application, and *dynamic*, where components can be manipulated during runtime. Despite the techniques named before, it is important to know the how, when and where, composition takes place in order to design a proper adaptive solution [16].

2.3 Self-Adaptive Software System Approaches

This subsection highlights some approaches related to self-adaptive software systems based on feedback loops, components, and even hybrid architectures, and communication standards that ground some of the structural and behavioral basis of this proposal.

2.3.1 Kramer and Magee's Three Layer Architecture

Kramer and Magee proposed a three level architectural model for self-managed systems [15]. The three layers are: *Component control, Change Management* and *Goal Management* (See Figure 2.1).

- *Component Control*
  This layer is responsible for the interconnection among components to accomplish the function

of the system. It facilitates the information about the status of the components. Furthermore, in this layer is where occurs the deployment, undeployment and interconnection of the system components.

- *Change Management*
  The second layer in this model is responsible for the execution of plans, that is, the actions to address context changes that violate the satisfaction of system requirements. The actions taken in this layer might include the introduction of new components, changes in the interconnection or even in the operational parameters.

- *Goal Management*
  At the highest level, the plans needed from the layer below are produced. Also, in this layer the introduction of new goals take place. These new goals are being introduced in the system in consequence of the changing environment that could require a different behavior of the system to achieve its goals or even changing the goals themselves.

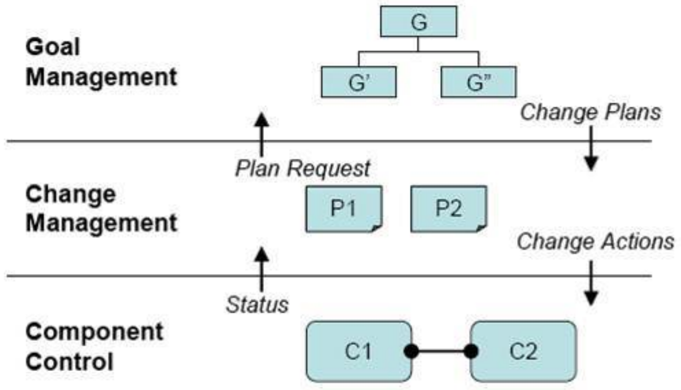


Figure 2.1: *Three Layer Architecture Model for Self-management [15]*

2.3.2 Autonomic Computing Reference Architecture (ACRA)

IBM proposed an architectural blueprint for autonomic computing [12]. According to this proposal, the management of a system involves four common activities: *collect information, analyze it to determine actions that need to be taken, create a plan with those actions*, and finally, *execute the plan*. In light of this, for the system to manage itself this process, the following conditions must exist:

- The management tasks known as: configuring, healing, optimizing and protecting, must be automated.
- These processes must be possibly initialized based on situations observed in the context.
- The system must possess enough knowledge to perform the automated tasks.

The IBM's proposal presents a five layer scheme, shown in Figure 2.2, to control self-management systems. However, the most relevant aspects in this proposal for our project are in layers 2 and 3 [12]:

- *Level 2 - Touchpoints*
  These are the components responsible to interact with the managed resources. The interaction refers to the awareness of the state and management of the resource trough two components: sensor and effector. The sensor is in charge of knowing the state of the resource either by acknowledging the properties and looking for changes, or by the events triggered over the changes in the resource. Meanwhile, the effector is in charge to notify changes over the resource either by the properties or by operations that are implemented in the managed element.

- *Level 3 - Touchpoint autonomic managers*
  These components implement the *intelligent control loop* over the managed resources, arranged typically in four scopes: single resource, homogeneous, heterogeneous and business system.

  This control loop is also known as the MAPE-K-loop: Monitor, Analyzer, Planner, and Executer (See Figure 2.3). These four parts communicate and collaborate to each other and exchange appropriate knowledge and information. The Monitor collects the information from the managed resources. The Analyzer observes the reported situation, and determines if any change needs to be made. The Planner creates the plan to achieve the changes, and the Executer provides the mechanism to perform the changes over the managed system.

  The common part of the MAPE-K loop, is the knowledge source, which is shared data among the autonomic managers in the system. In order to do that, the knowledge must by expressed using common syntax and semantic to the system. There are at least three ways to specify knowledge: through policies, activities inside the autonomic system or actions inside the autonomic control loop.

2.3.3 Feedback Control Loops

As mentioned previously, self-adaptive systems decide autonomously about the changes the system needs to perform. In order to take those decisions, the system must be aware of the context and of itself. Hence, the system must know the output of its actions and use them as feedback to take the correct decision to be made [17]. Control theory is about knowing the characteristics of the system. It deals with understanding how the desired input is affected by disturbance and noise reflected over the measured output. In computing systems, even though control theory is necessary to ensure that systems have desired output characteristics all over its operation it is very difficult to apply it mainly because computing systems are non-linear in general [11].

Figure 2.4 shows a SISO[1] feedback control system, whose elements are described as follow [11]:

- *Reference input*: This is the desired control objectives to be achieved. It is a reference used by the control system to check if the objectives are being reached.

[1]SISO: Single-Input, Single-Output

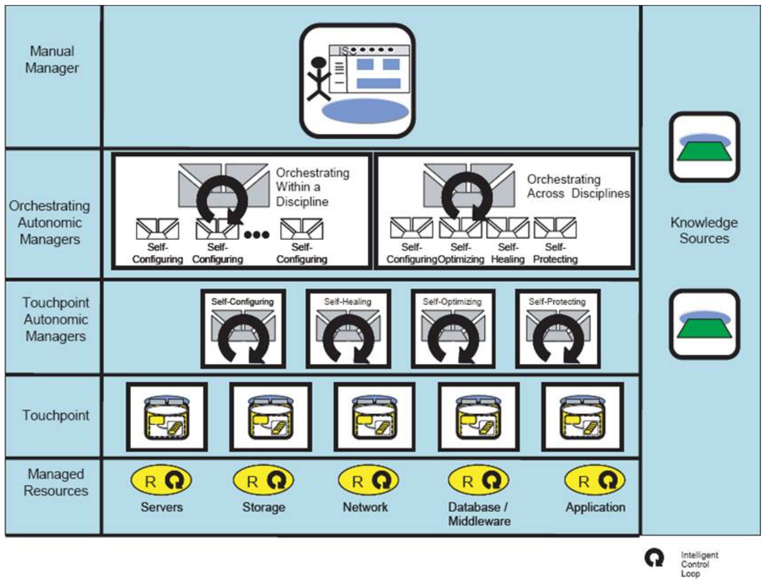


Figure 2.2: *Autonomic computing reference architecture (ACRA) [12]*

- *Control error*: The result of the comparison between measured output and the reference input. In other words, it expresses how differs the system from the desired goal.
- *Target System*: The system to be controlled.
- *Controller*: This element is responsible for computing the control input for the target system to achieve the reference input.
- *Control input*: The information dynamically produced by the controller to affect the behavior of the target system.
- *Disturbance and Noise*: These two elements correspond to context sources. Disturbance is any changes in the external environment that affect the target system. Noise is any effect inside the target system that affects the measured outputs.
- *Measured Output*: The measurements being monitored in the Target System.
- *Transducer*: This is an optional element. Transducers transform the measured output to the measurement units of the reference input, so they can be compared.

Feedback loops are important for this project because they provide the generic mechanism required for self-adaptation. Feedback loops are important in software processes to manage and support software evolution [18]. The generic feedback loop (Figure 2.4) represents the four main activities required for self-adaptation: First the monitor probes *collect* the relevant data through sensors to reflect the current state of the system. Second, the analyzer element *analyzes* the collected data and searches for symptoms about the desired behavior and the current state. Third, the planner element *decides* whether it is necessary to adapt the system to reach the control objectives. And last, the executor element *instruments* the adaptation actions through effectors [9]. The steps in the feedback loop cycle are very similar to the tasks of autonomic computing.

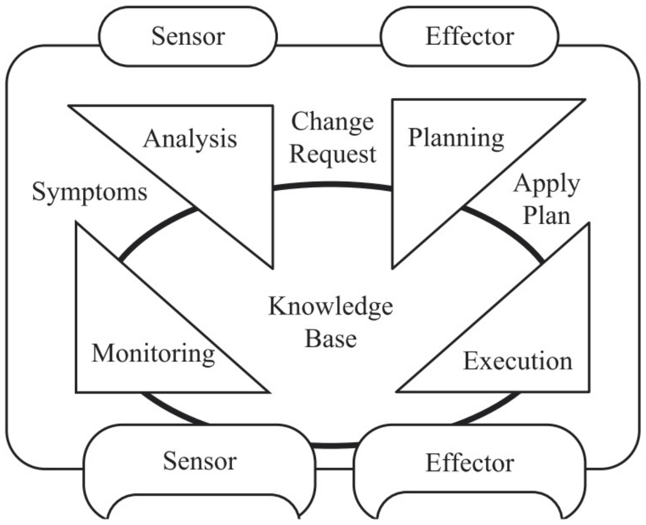


Figure 2.3: *Functional arrangements of the Autonomic manager (ACRA) [12]*

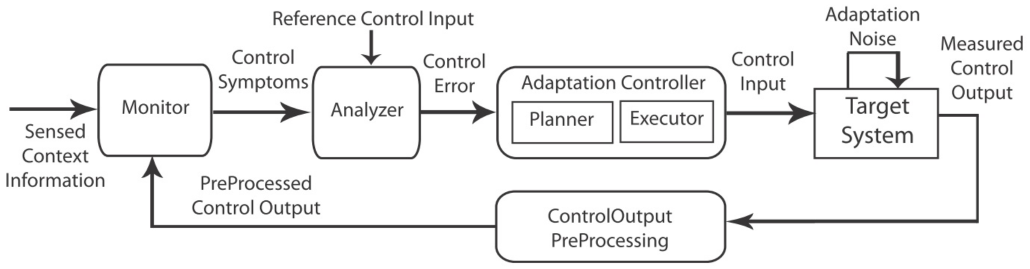


Figure 2.4: *SISO Feedback control block diagram with explicit functional elements and corresponding interactions to control dynamic adaptation in a software system [30]*

The main reason to implement feedback loops into computing systems is to reduce the effects of uncertainty that appear in different forms as a consequence of the shifting context and the changes in business goals. Therefore, it is necessary to make explicit feedback loops in the design of autonomic systems in order to reason about its behavior and to analyze the adaptation mechanism separately from the target system operation.

2.3.4 DYNAMICO: A Reference Model for Context-Based Self-Adaptation

Undoubtedly, in order to implement adaptation through feedback loops it is necessary to make explicit the management of self-adaptive properties as the control reference goals, and the separation of concerns by using different feedback loops to achieve the reference goals [30].

The reference model for context-based self-adaptation by Villegas *et al.* illustrated in Figure 2.5 is based on the SISO feedback control (Figure 2.4) with explicit functional elements and corresponding interactions to control dynamic adaptation. These are the MAPE-K[2] loop elements.

[2]MAPE-K: Monitor, Analyzer, Planner, Executor and Knowledge Base [12]

The separation of concerns is a key aspect in this model, which defines three subsystems to achieve self-adaptation [30]:

1. *Control objective manager:* Manages the target system's purpose in terms of its control objectives, according to the policies given by administrators. These control objectives can change at runtime according to user-level negotiations. However, this management implies to express quality attributes quantitatively and have them updated at runtime.

2. *Context manager:* Responsible for maintaining the pertinence and relevance of the context monitoring infrastructure with respect to the target system under changing conditions of execution. In order to do that, the context manager must be able to decide about the facts, past, present and future states of the context variables of interest.

3. *Adaptation mechanism:* Responsible for the adaptive actions over the target system according to the evaluation of its behavior. That means, to guarantee the accomplishment of the target system's purpose to satisfy its quality attributes.

Of particular importance in this reference model are the interactions between the three loops. Even though they are designed independently they require to operate together to achieve the system objectives.

2.4 Component-Based Software Engineering

As defined by Councill *et al*, a software component is a software element that can be independently deployed and composed in any other model according to a standard composition. The difference between a software component and any other software element is the use given by components in the matter of quality features [10]. Moreover, software components hide all the complexity of the software implementation by only exposing the services that the component requires and provides through its interfaces. This property allows to assemble software pieces depending only on the services and not depending on the inside implementation, the number of classes, the programming language. This easy way of assembling components allows software engineers to reuse, modify or even extend software without the difficulty of development maintenance because as software components are independent from each other, they can be easily replaced [23].

For this proposal, a separation between the adaptation mechanism and the software system is required in order to analyze and maintain each system as its own. Component-based software allow separation but also connection among components through interfaces. Also, a component-based design will provide flexibility and extensibility and the possibility of operation between software systems running on different platforms and written possibly in different programming languages.

2.5 Software Architecture, Reference Models and Reference Architectures

According to Len Bass *et al.* an architecture in software defines the elements that compose the system and the relationship between those elements, and it hides the information that is not relevant for the interaction among them [1]. The individual behavior of each element is an important part of the architecture which allows the appropriate communication between them.

The design of a reference architecture follows different stages where each stage represent architectural decisions and architectural binding choices. Figure 2.6 shows the relationship among the design elements needed to build a reference architecture.

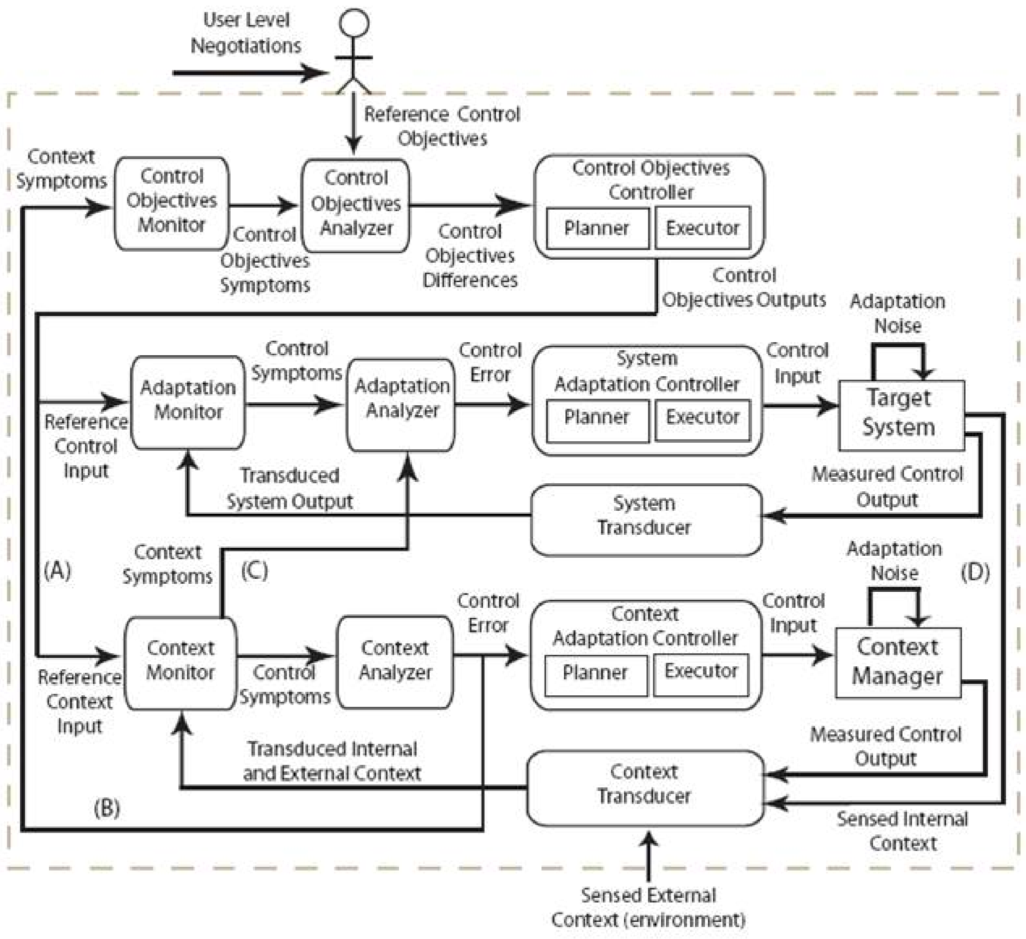


Figure 2.5: *Reference model for context-based self-adaptive systems [30]*

2.5.1 Reference Model

A *reference model* represents the decomposition of a well known type of problem in several parts that work together to solve the problem. This decomposition is standard in certain knowledge area and it clearly specifies the parts and the functions inside the model [1]. Feedback loops, for instance, is a reference model in control theory, and the 7-layers OSI is the reference model for communication systems.

2.5.2 Architectural Patterns and Architectural Styles

An *architectural pattern* is described as a set of elements in component types and the relationships among them. Moreover, it exhibits known quality attributes such as performance, security or availability. A set of architectural patterns is known as an *architectural style*. As an addition, an architectural style also defines a pattern for the runtime control and data transfer of the component types but is not an architecture [1] . The main difference between an architectural style and a reference model, is that the second one is well-known and accepted by a community.

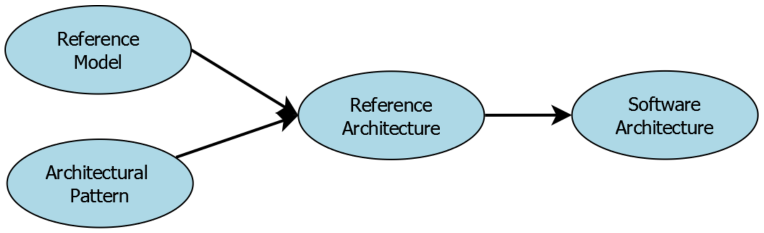


Figure 2.6: *Reference Architecture derivation relationship between reference model, architectural patterns, reference architectures, and software architectures. (The arrows indicates that subsequent concepts contains more design elements)[1]*

David Garlan and Mary Shaw considered a number of common architectural styles used in many systems and the combinations of those in a single design [8], some of those architectural styles are necessary to design the reference architecture for this case of study and it will be described in Chapter 3.

2.5.3 Reference Architecture

A reference architecture is a mapping between a reference model and a set of software elements with the data flows among them. Each element in the reference architecture implements totally or partially a function previously identified in the reference model[1].

Chapter 3 details the reference architecture for self-adaptive systems proposed in this work.

Chapter 3

A Reference Architecture for Component-Based Self-Adaptive Software Systems

As mentioned in Chapter 2, a reference architecture is the result of the mapping between a reference model and the software elements of a given design and corresponding data flow. This chapter presents our reference architecture for designing and implementing component-based self-adaptive software systems. We explain how components of this architecture are built and the role of each component in this architecture and explain the data flow required to connect the components in this architecture.

3.1 A Reference Model for Context-Driven Self-Adaptation

The case of study of this project is based on the running example presented by Tamura *et al.* [24] called the Reliable videoconference System (RVCS). In summary, this RVCS is a self-adaptive software system with a server that hosts videoconferences and mobile users which register and attend the conferences.

In this scenario two software elements interact: an *end-user application* in a mobile device and a *server application*. Figure 3.1 shows the main three functionalities of the system to be modeled: (1) the end-user application allows the user to register in the videoconference system, (2) the application allows the user attend to a particular videoconference in execution, and (3) a server application in the videoconference system establishes the service conditions, also known as SLO[1], with the end-user application. The end-user application itself also establishes the conditions with the server, that is why the arrow is bi-directional.

It is worth nothing in this scenario that both applications (end-user and server) must perform a set of activities to ensure the proper behavior of the entire system, that means the need of implementing self-adaptation. The IBM's proposal for adaptive systems includes an *intelligent control loop*[12]. This control loop is also known as the MAPE-loop: Monitor, Analyzer, Planner, and Executer. These four parts communicate and collaborate to each other and exchange appropriate knowledge and information. The Monitor collects the information from the managed resources. The Analyzer observes the reported situation, and determines if an adaptation is required. The Planner creates the plan to realize the required changes, and the Executer provides the mechanism

[1]SLO (Service Level Objective) is an element of the SLA (Service Level Agreement) between a service provider and a service consumer. The SLO measures the achievement of a quality attribute of the service.

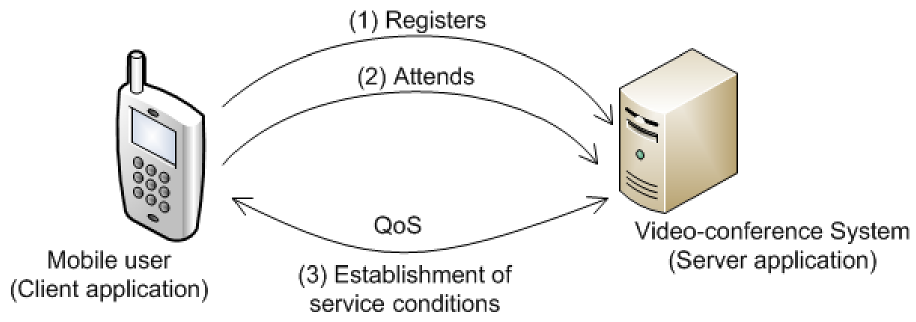

(1) Registers
(2) Attends
QoS
(3) Establishment of service conditions
Mobile user (Client application)
Video-conference System (Server application)

Figure 3.1: *Basic functional diagram for the case of study. (The arrows point the direction from source to target in the corresponding action)*

to instrument the changes over the managed system. Figure 2.3 illustrates the IBM's autonomic manager.

To guarantee the extra functional requirements of the system in the case of study, it is required for the system to perform a self-adaptation, if needed, in order to meet the objectives of the system. Table 3.1 shows the correlation between the MAPE activities and the case of study elements, in order to meet the desired behavior of the system.

The complete details of the Case of Study for this project are described in Section 4.2, page 31.

MAPE	**Case of study elements**	
	User *(Mobile Application)*	**Videoconference System** *(Server Application)*
Monitor	- Type of connection and bandwidth restrictions (Throughput and Availability) - Point of access, internal or external (Confidentiality)	- Amount of registration request in a specific interval of time (Throughput) - Amount of time during registration (Performance and Capacity) - Amount of users attending a videoconference in a specific interval of time (Availability) - Amount of videoconferences in execution at the same time (Throughput)
Analyzer	-SLO's fulfillment -Need for adaptation	
Planner	-Adaptation plan to achieve the SLOs	
Executer	-Execute the adaptation plan in the target system	

Table 3.1: *Specific actions for each element in the system in the Case of study*

3.2 Architectural Patterns in Context-Driven Self-Adaptation

As mentioned in Section 2.5.2, David Garlan and Mary Shaw considered a number of common architectural styles. For this reference architecture, three of these styles are reused.

Pipes and Filters

In this architectural style, each component has a set of inputs and a set of outputs, and the data is processed in such a way that both streams are different. The data processing is accomplished by applying local transformations [8].

The pipe and filter elements are described as follow:

- FILTER: This component uses the data in the input stream, applies transformations to it and deliver it through the output stream. Filters are unaware of their predecessor or successor filter in the line.
- PIPE: This component is the connector of the filters and transport the data between them.

As a consideration, the basic premise of this architectural style is that the order of the filters is not important. However, this consideration does not apply for the reference architecture to design in this project, because the order of the filters is important and it can't be altered. This specialization is called *pipeline*, which restrict the topology to a linear sequence of filters, the amount of data on a pipe, and a well-defined type of data [8].

In our reference architecture we have the necessity to transform data from one component to another. This is, the data that is being monitored in one element needs to be analyzed later by other element, but as this data is not of the same kind, it needs to be transformed from one element to the next. The same consideration applies for the analyzed data and the development of the adaptation plan.

Event-based

This architectural style is used in solutions in which a software component provides a set of procedures and functions to interact with other components by invoking those routines. However, the implicit invocation implied by the association of these routines to the foreseeable events that can be faced by the system at runtime is currently the most used technique to manage context event processing. Those events are associated to one or more registered procedures, in that case the system implicitly invokes the corresponding procedures to attend the event [8].

The event-based elements are described as follow:

- COMPONENT: This element is a module whose interface provides a set of procedures and a set of events. The component can associate some of its procedures with events of the system.
- PROCEDURE: This is the piece of operation that responds to a triggered event. Is bound with an event, and it is invoked at runtime implicitly.

In our case of study we need to execute and/or trigger event over some components such as the target system and the sensors and effectors.

Blackboard

This architectural style is the type of a repository. In this style a component is responsible to store data and share it with other components [8]. The blackboard elements are described as follow:

- KNOWLEDGE SOURCES: This is the element with the knowledge that will be store in the blackboard data structure. Are independent to each other and only interact through the blackboard.
- DATA STRUCTURE: This is the stored data needed to solve a problem. The knowledge sources make changes to the backboard to reach the solution to the problem. This data is shared among the knowledge data sources.
- CONTROL: This element alerts the knowledge sources when a change in the blackboard has been made.

In our case of study we need to exchange data information among the elements regarded to the adaptation requirement.

3.3 Reference Architecture Data Flow

In Section 2.5, we described the process to derive a reference architecture. This reference architecture is the result of the mapping between the reference model and the software elements along with the data flow required to connect the components.

3.3.1 Context Metamodel

According to MDE[2] a metamodel describes concepts that can be used from modeling the model, and it describes in an abstract way, the possible structure of models. The keyword on modeling is the abstraction of real scenarios. Metamodels and models have a conformance relationship [22].

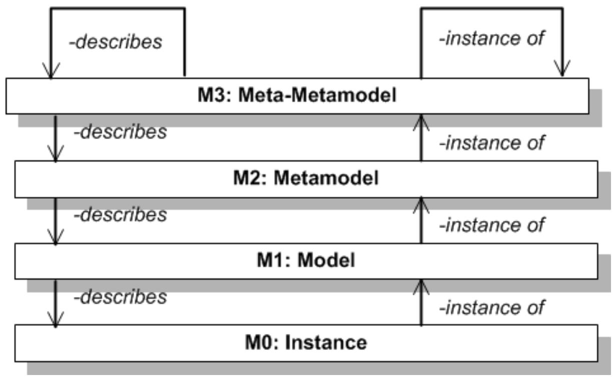


Figure 3.2: *The four metalevels of OMG [22]*

OMG[3] defines four metalevels for MDE as shown in Figure 3.2. The lowest level M0, is the final an concrete level or for better understanding, the runtime program. The model for this level,

[2]MDE: Model-Driven Engineering
[3]OMG: Object Management Group

placed in level M1, defines the classes with their attributes, relationships and constrains. In M0, realized as classes in a programming language such as Java, the class instances take values that represent concrete snapshots of the system at runtime. Hence, M1 is the first abstraction of M0. Moreover, M0 is an instance of M1 but not necessarily the only one. Following this scheme, The metamodel level M2 is an abstraction of a model M1. This metamodel defines the structure which M1 is built. The highest level is M3 which represents the abstraction of the metamodel. This last level is usually to define modeling languages (i.e. The MOF[4] is the OMG's meta-metamodel to define M2 languages such as UML).

For this reference architecture's data flow, the context model is based on the context metamodel proposed by Villegas *et al.* [29] shown in Figure 3.3. For our reference architecture, not all the metaclasses of that context metamodel are necessary, given the scope of this project.

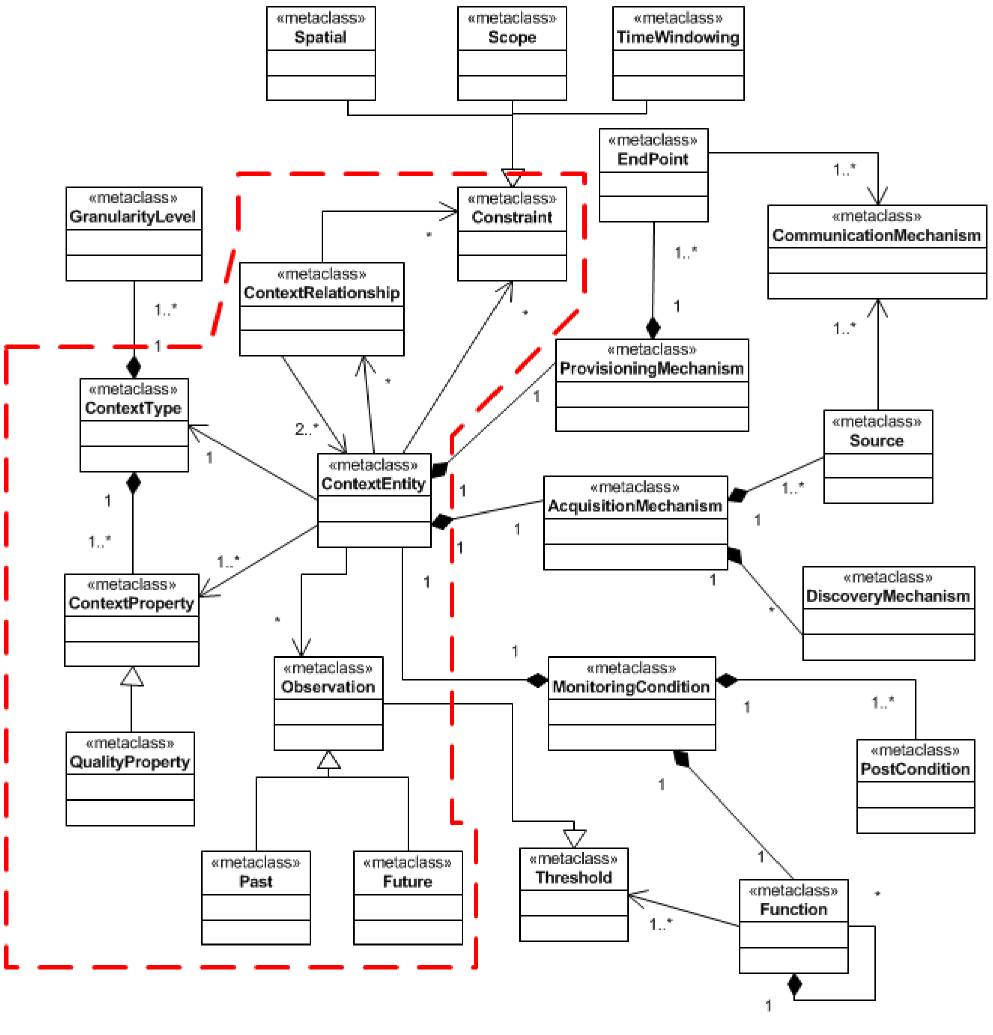

«metaclass» Spatial
«metaclass» Scope
«metaclass» TimeWindowing
«metaclass» EndPoint
«metaclass» CommunicationMechanism
«metaclass» GranularityLevel
«metaclass» Constraint
«metaclass» ContextRelationship
«metaclass» ProvisioningMechanism
«metaclass» ContextType
«metaclass» ContextEntity
«metaclass» Source
«metaclass» AcquisitionMechanism
«metaclass» DiscoveryMechanism
«metaclass» ContextProperty
«metaclass» Observation
«metaclass» MonitoringCondition
«metaclass» PostCondition
«metaclass» QualityProperty
«metaclass» Past
«metaclass» Future
«metaclass» Threshold
«metaclass» Function

Figure 3.3: *Context Metamodel by Villegas et al. [29] and the subset we used*

[4]MOF: Meta Object Facility

3.3.2 Context Model

As mentioned before, a model is an instance of a metamodel, in our proposal we use a subset of the context metamodel proposed by Villegas *et al.* shown in Figure 3.3 grouped by the red dashed line [29] as a model for the data to be exchange among the components of our reference architecture. This subset excludes the acquisition mechanism for the context and the provisioning mechanism because of the scope of this project.

Context Model entities

As noticed in the IBM's proposal explained in Section 2.3.2 the MAPE components are connected to each other as a mechanism for adaptation. Each component receives information about the context, process it and deliver it to the next components in the loop. Figure 3.4 shows the given names for the data to communicate each component in the MAPE loop: *Context Data, Context Event Information, Diagnosis, Adaptation Actions* and *Commands.*

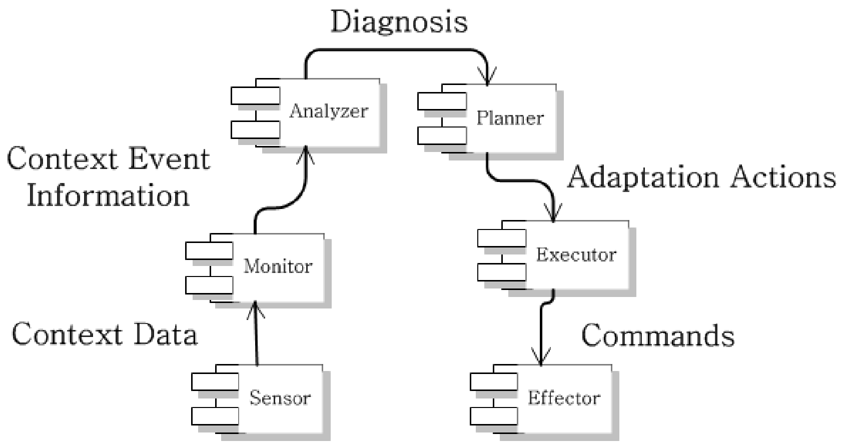


Figure 3.4: *The MAPE loop data*

However, only the *context data, context-event information* and *diagnosis* are instances of the context model. The data *adaptation actions* and *commands* are not instances of the context model but sets of strings.

Considering the above, the context model required for the reference architecture data flow must represent the three types of data mentioned before: *context data, context-event information* and *diagnosis.* Figure 3.5 shows the context model derived from the context metamodel subset to express the MAPE loop data flow which is related to the context. In the same way, the model is transformed into an instance, in this case a class diagram to represent the context. Figure 3.6 shows the correspondence from the model to each class that represent the context. See Appendix B.1 for full details on the corresponding Java implementation Java implementation.

However, to illustrate this MAPE loop data required to communicate the MAPE loop components, and the relation with the context model, lets assume the following scenario: *An SLO defines that for the videoconference application the time during the process of a client registration must be less or equal than* $\boldsymbol{x}$ *seconds. The SLO also requires that the monitoring for the registration time must be in periods of 1 minute each in order to analyze the average time of all registrations during*

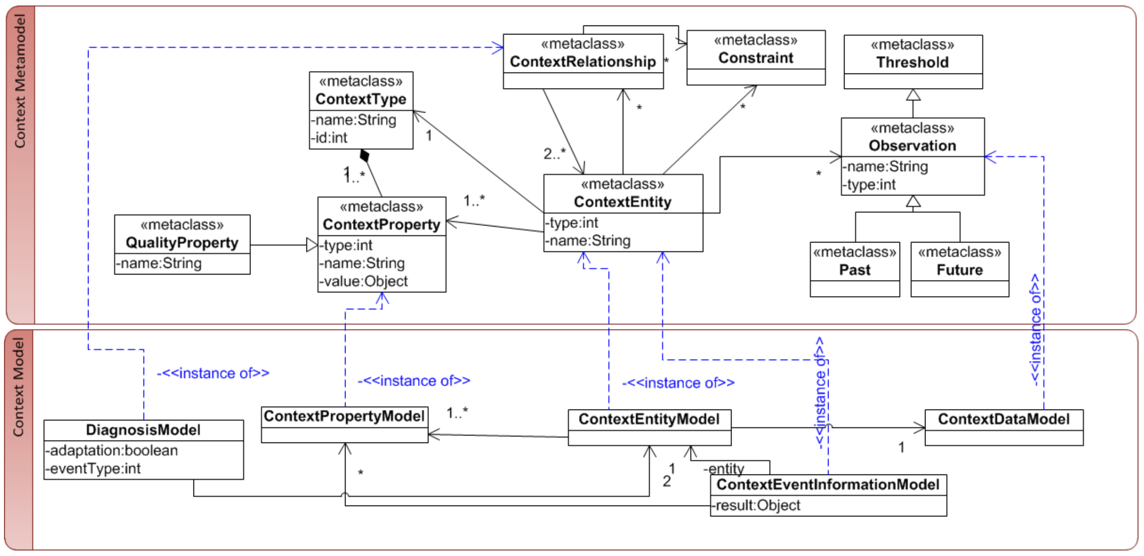


Figure 3.5: *Context Metamodel vs. Context Model (Dashed arrows represent the "instance of" relationship)*

that period of time. If during that minute the 80% of requests compute an average registration time above the x seconds then adaptation is required because the SLO is not being fulfilled.

Context Data: this type of data is the result of the Sensor component activity (*See Figure. 3.4*). This type of data carries the information of the context events being monitored. Hence, this data must represent the context entity that is being sensed along with its set of properties, and the property that triggered an event that was sensed or the property that is configured to be sensed. Indeed, the context model entity Context Data is an instance of the Observation metaclass. This Observation belongs to a Context Entity and its set of Context Property objects.

For our example scenario, the Context Data will indicate that the observation has been made after some registration request. The registration request is the Context Entity and the Context Property is the time that the registration took.

Context Event Information: Once the Monitor component receives the Context Data from the Sensor component and processes it, the Monitor must deliver substantial information about the current state of the context. The Context Event Information data, must represent the current state of a context entity according the SLO terms by the performance of a set of operations. In those terms, an Context Event Information data is an instance of the ContextEntity metaclass and its set of Context Property instances of ContextProperty metaclass to represent the current state.

In our example scenario, the Monitor component usually receives more that one Context Data about registration time. According to its responsibility, the Monitor must inform all of the Context Data during a periods of 1 minute, lets assume 115 in total. Once the period of 1 minute is reached, the Monitor would compute the result of its operation, that means, the total percentage of registration duration times above the **x** seconds, that is 85%. Then, the Monitor component delivers to the Analyzer the registration times as a Context Entity and a set of properties: the percentage result and the amount of registration requests.

Diagnosis: Finally the Analyzer receives the Context Event Information from the Monitor and evaluates the fulfillment of the SLO. During this evaluation, the Analyzer component must deliver a Diagnosis data as a relationship between two context entities: the context entity in its *current*

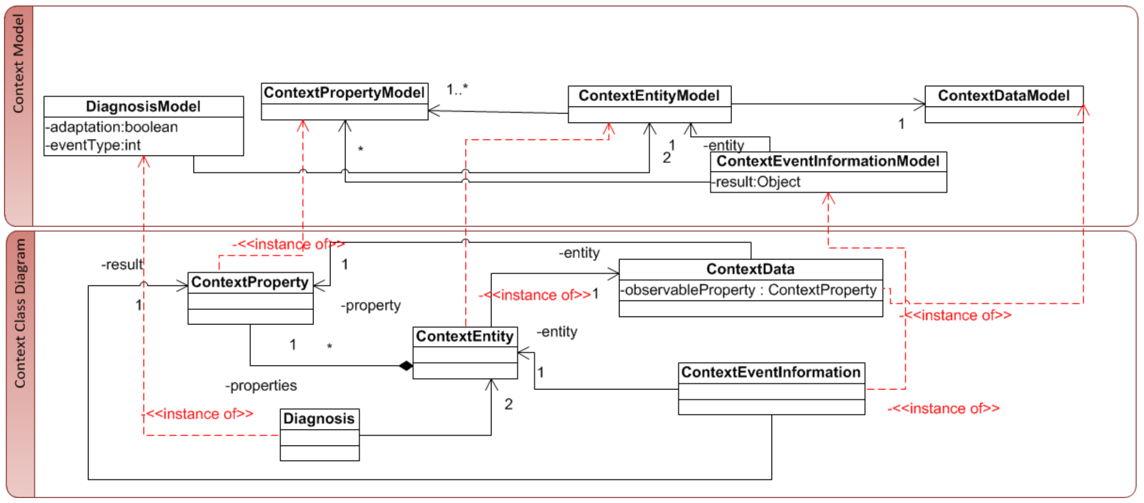


Figure 3.6: *Context Model vs. Context Class Diagram (Dashed arrows represent the "instance of" relationship)*

state and the same context entity in its *desired state.*

Following this, for the described scenario the Analyzer will compare the Context Event Information received with the SLO terms and determine that is not being fulfilled. Then the Analyzer delivers to the Planner the Diagnosis data where the context entity 'registration request' has a current property 'duration time' with value 85% and the desired value for the same context entity in the property 'duration time' must be 80%.

At this point, the Planner receives the Diagnosis and continues the loop for the adaptation mechanism, as described in the Figure 3.4.

3.3.3 Context Type Tree

The adaptation mechanism with some human assistance, must perform a classification of the context entities that is able to identify. We will consider this classification as a *context type.* According to the Context Metamodel in Section 3.3.1, context types are instances of the ContextType metaclass and in the model it becomes an unique attribute. Following the relations in the context metamodel, this *type* attribute belongs to three elements in the model: *Property, ContextEntity* and *Context Event Information.*

Figure 3.7 shows the tree for the different values of ContextType that are possible in our reference architecture. However, this is not an static tree, because in a concrete architecture this information will depend on the context reality. Hence, this context type tree belongs to the scenario for the case of study but can serve for other applications.

3.4 Component-Based Reference Architecture Proposal

As described in Section 2.5 IBM's proposal for adaptive systems includes an intelligent loop called the MAPE loop [12]. One of the main goals of this project is the development of an architecture for component-based self-adaptive systems that is extensible and with the level of abstraction required

- Context types
  - (1) Information
    - (1) Data Property
      - (1) String
      - (2) Int
      - (3) Double
    - (2) Occurrence
    - (3) Operation
      - (1) Sum
      - (2) Count
  - (2) Event
    - (1) Invoke
    - (2) Occurrence
  - (3) Source
    - (1) Hardware
      - (1) Hardware Service
      - (2) Device
      - (3) Configuration Value
    - (2) Software
      - (1) Method
      - (2) Attribute or Property
      - (3) Software Service
  - (4) Execution Time
    - (1) After
    - (2) Before
    - (3) During

Figure 3.7: *Context Types tree*

to implement self-adaptive software. Figure 3.8 presents our reference architecture in which the MAPE loop is included based on the original IBM's proposal.

To clarify the understanding of this architecture, the components are divided in three groups:

1. The components related to the **interaction with the context**: The Target System components and the Context-Entities.

2. The component related to allow the **human interaction** in order to administrate the adaptation mechanism: the Administration Management Console (AMC) component.

3. The components related to the **adaptation mechanism**: the Sensor, the Monitor, the Analyzer, the Planner, the Executor, the Effector and the Knowledge Base *(Also known as MAPE-K)*.

3.4.1 The Context Interaction Components

As defined by Villegas *et al.* **context** is anything that interacts with a target system and it can be affected by the target system or the same way round [29]. The components in this group are those related to the context, either because they are the context or affected by the context.

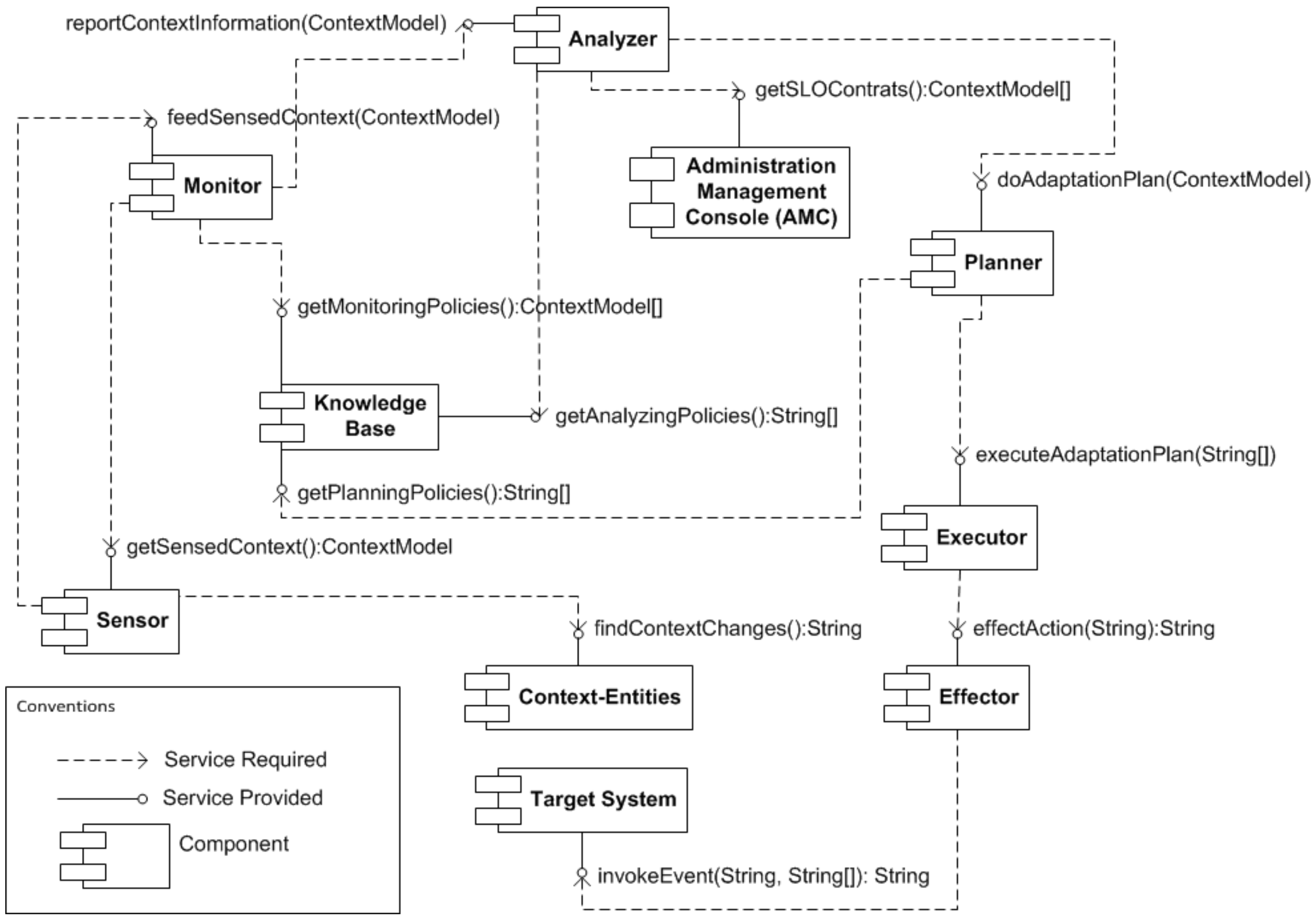

reportContextInformation(ContextModel)
Analyzer
feedSensedContext(ContextModel)
getSLOContrats():ContextModel[]
Monitor
Administration Management Console (AMC)
doAdaptationPlan(ContextModel)
Planner
getMonitoringPolicies():ContextModel[]
Knowledge Base
getAnalyzingPolicies():String[]
executeAdaptationPlan(String[])
getPlanningPolicies():String[]
Executor
getSensedContext():ContextModel
Sensor
findContextChanges():String
effectAction(String):String
Conventions
Service Required
Service Provided
Component
Context-Entities
Effector
Target System
invokeEvent(String, String[]): String

Figure 3.8: *Component-based reference architecture for self-adaptive systems*

Target System

In the IBM's proposal, this element is also called the *managed element* [12]. In our proposal, the Target System comprises a set of components where the adaptation is going to take place. The Target System is the system we intend to adapt and must be a component-based software system executed in a component runtime system. Moreover, the specific services to instrument the adaptation actions are provided by the runtime and not the target system. The monitoring services must be provided by components deployed by the adaptation strategy.

The Target System in our reference architecture is represented as one component, however, this does not imply that is smaller or less complex than the adaptation loop.

Context-Entities

These components represent the context monitored and whose behavior will affect the target system determining its adaptation. In some cases, the Context Component and the Target System Component are the same, because the system itself is its own context.

3.4.2 The Human Interaction Components

This reference architecture is based on the self-adaptive principle and in order to perform this self-adaptation some human intervention is required. This component allows human interaction to define policies and configurations over the adaptation mechanism.

Administrator Management Console (AMC)

This component allows system administrators to configure the SLOs that the Target System must fulfill during execution and that the adaptation mechanism must evaluate to decide if adaptation is required. This component follows a SLO's general definition of the *Quality of Service (QoS) Contracts* based on the e-graph representation proposed by Tamura *et al.* [26].

To illustrate the AMC role in this reference architecture, lets assume an hypothetical SLO definition in this scenario that indicates that *the videoconference server application must attend no more than one hundred (100) client's connection requests per server in an interval of one (1) minute, and if that number of requests is reached, another server must be turned on and the client requests must be balanced between all the online servers.*

The system administrator configures in the ACM component these two Context-Entities, the *number of servers* and the *number of connection requests*, are part of a QoS Contract meant to be fulfilled by the Target System. Also, for those context-entities will specify acceptable values, *i.e.* less or equal that 100 for the connection requests and 1 for the number of servers online. Finally, the Adaptation Mechanism will use this SLO information to meet the adaptation requirements.

3.4.3 The Adaptation Mechanism Components

These components are derived from the original IBM's MAPE-K proposal [12] and as described before, they are in charge of the adaptation process. This set of components process data information about the context into adaptation decisions and therefore is the heart of this reference architecture.

Sensor

This component is the starting point of the MAPE-K loop and it is configured to make measurements about the context with the purpose of sensing a specific variable of interest during runtime. A sensor *(also known as Monitor Probes)* acts as a watcher and interacts with the monitor component as shown in the Figure 3.8. The sensor provides a representation of the target system's state in a specific variable of interest and other external systems information, in the form of context entities needed for the adaptation mechanism.

This component has two action modes: *Push* and *Pull*. In the Push mode, the monitor sends the order to perform sensing over the context. This mode is useful in the case the monitor needs information of the context in an specific time. In the Pull mode, the sensor performs the sensing over the context every time it detects expected changes or events triggered in the context and sends it to the monitor component as information that *something* happened in the context.

Monitor

As described by IBM, this element is responsible to monitor the sensed context [12]. The data required to perform this monitoring is called *context data* and is provided by the monitor probes, in this case, the Sensor component. However, this context data is every information about the context that the monitor probes can sense, either because something happened in the context or because the Monitor asked it to gather information in an specific time. Despite this, all the context data that the monitor probes are providing are not necessarily information required to be monitored. It is the Monitor's responsibility to filter all that context data to keep only the one that is relevant. To filter this information, the Monitor must configure some *Monitoring Policies* that must be derived from the SLO definitions. In this reference architecture proposal these monitoring policies are provided by the *Knowledge Base Component*.

As an example, lets assume that the context to be sensed is the one affecting the videoconference server application. The monitor probes will notice every change in the server software system, either hardware (change of smartphone geographical location, change of network access, etc) or software (the number of clients connected to the videoconference server, the service of registration is up, etc) and will announce to the monitor that those events of change had occurred.

To fulfill our hypothetical SLO, the resulting Monitoring Policy will set that the context entities to be monitored are *the client's requests of connection* and *the number or servers online* in a period of one (1) minute. According to this case, the Monitor will filter all the context data to only the one that inform about the client's requests of connection to the videoconference server and the number of servers online attending those requests. With this context data, the Monitor must build an *Context Event Information* to report the context events to the Analyzer component.

Analyzer

According to IBM's proposal, this element is responsible to study the *Context Event Information* given by the Monitor in order to evaluate if the system objectives are being fulfilled [12], in this case, the SLO. Moreover, the Analyzer's objective is to determine the achievement of the SLO, but also it has to decide whether an adaptation is required. As an income, the Analyzer receives from the monitor the context events and then the Analyzer must evaluate it against the SLOs configured in the system. To perform this evaluation the Analyzer must configure some *Analyzing Policies* provided by the *Knowledge Base Component* that determine the analyzing parameters to decide the fulfillment of an SLO.

According to our scenario, lets assume that the Monitor gives to the Analyzer the Context Event Information of one hundred ten (110) clients in an interval of one (1) minute and one (1) server is online. The Analyzer must retrieve the SLO related to that context entity and evaluate the level of fulfillment. In our case the result can be either that it is below, above or acceptable as the expected. In our example of SLO, the maximum of clients per minute per server is one hundred (100) which means that this result is above the expected. In this case, the Analyzer must build a *Diagnosis* for the Planner in order to inform the adaptation requirement by comparing both context states: *current* and *desired*. The current state is one hundred ten (110) clients per minute attended by one server, and the desired is one hundred (100) clients per minute attended by one server according the SLO definition. This difference in the diagnosis means that the SLO is not being fulfilled and adaptation is required.

Planner

As described in the IBM's MAPE-K loop proposal, the Planner element is responsible lo plan the required adaptation over the system[12]. This element is responsible to build a plan with all the *adaptation actions* needed in order to perform certain changes in the target system and then, reach the desired state of the context entity provided by the Analyzer in the diagnosis. This plan is a full step-by-step set of instructions that can be delivered as a set of Strings, and that must be processable by the Executor. To build this plan, the Planner must follow some *Planning Policies* provided by the *Knowledge Base Component*.

Following our scenario, the adaptation plan might include the following actions: (1) Find an available server for the videoconference service. (2) Send the "turn on" command. (3) Start the videoconference service (4) Configure the connection balance as true. These actions must be set as an instruction and that instruction will depend of the executor programming language. Finally, the plan built by adaptation actions, is delivered to the Executer to close the loop of the adaptation mechanism.

Executer

This is the last element in the IBM's intelligent Loop and is responsible to execute the adaptation plan[12]. This component has the purpose of translating to commands every step of the plan and guarantee the correct execution over the target system. This component is connected to the Effector, the component that can instrument directly the target system.

Effector

This element is the finishing point of the MAPE-K loop interfacing the adaptation mechanism and the target system. The Effector component is configured to effect the changes needed to alter the target system's behavior according to the adaptation needs. The Effector can either modify parameters of system operations or execute operations. This component is connected to the Executor component and receives from it the commands to execute over the target system.

Knowledge Base

In the IBM's proposal, the Knowledge element is connected to each of the four components in the MAPE-K loop[12]. As described in Chapter 2.5, the architectural style for this component is Blackboard and the main processing for this component is to store data. This component's main purpose is the provisioning of policies for other components in the MAPE-K loop (i.e. Monitoring, Analyzing, and Planning policies). Among other implementations, the knowledge base can be used to manage the historical behavior of this adaptation mechanism during time, but this application is out of the scope of this project.

Chapter 4

Reference Architecture Evaluation

In Chapter 3 we presented our reference architecture in detail and introduced an application scenario to illustrate the requirements that justify our design decisions. In this Chapter we recall this application scenario and use its implementation to evaluate a concrete architecture derived from our reference architecture. The goal of this evaluation is to determine the practical feasibility of our reference architecture. and explore its extensibility, in this case to support the dynamic adaptation of SCA Software applications.

4.1 Component-Based Reference Architecture Implementation

Several platforms for developing, deploying and executing component-based software are available, currently: JEE *(Java Enterprise Edition)* is the initiative proposed by *Sun*[1] by the implementation of EJB *(Entity Java Beans)* [13]. On the other hand, Microsoft has an initiative using its platform .Net for the development of software components but the specialty of this platform is software components for web applications [4]. However, the intention of our proposal is the extension and distribution capability and the contribution for the software engineering community, in which case an open source platform seems more appropriate than a licensed one. In order to support this initiative we chose Java J2SE *(Java Standard Edition)* platform because even though J2EE is meant to be for component-based software development it does not follow an standard for component-based software.

4.1.1 Service Component Architecture (SCA)

The SCA *(Service Component Architecture)* Specification was proposed by a group of vendors *(i.e. BEA, IBM, Oracle, SAP)* as a response to the need for component-based software development having on mind two main definitions: (i) a way to define components and (ii) a mechanism for describing how those components work together [19].

As described by Beisiegel [2], an SCA application can be built by more than one component each running on a different machine and even in a more complex scenario, written in different programming languages, relying on different communication mechanisms to operate properly. Indeed, the main goal for the SCA specification was to define the elements of these components and the interaction among them. As a result, *components* and *composites* are defined in the specification.

[1]Sun company it is now owned by Oracle company

Components are software applications with three elements: *services, references* and *properties* and **Composites** are larger structures that combine components. Composites are described by an XML-based file in a format called the Service Component Definition Language (SCDL)

In Figure 3.8 awe presented the abstract component of our reference architecture. In this figure, a service is the interface provided, a reference is a services required from another component and properties are not specified. Figure 4.1 shows an instantiation of our reference architecture in terms of SCA components for a specific implementation. In this Figure services *(Green chevrons)* and references *(Purple chevrons)* are explicit between components. It is important to note that not all the services provided and required in the reference architecture are described in this implementation as SCA services and references, the reason is because of the scope of this evaluation. In this case, for example there are no planning or analyzing policies, which means that this implementation uses an static evaluation of the SLO Contracts (Analyzer component) and there is no Adaptation Plan from the Planner component.

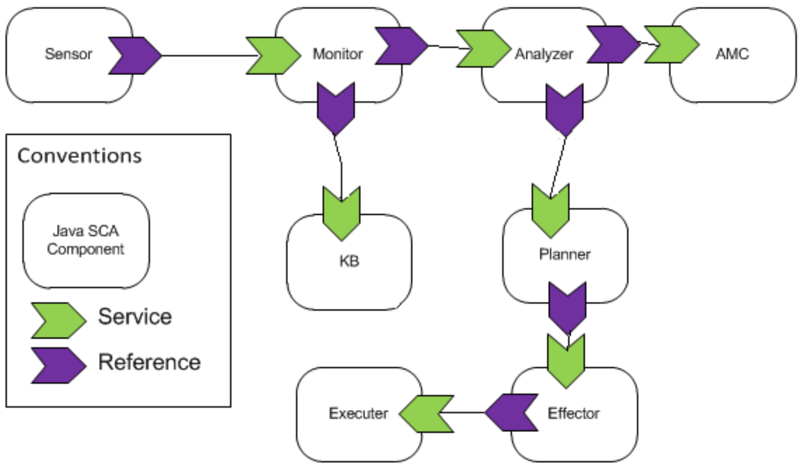


Figure 4.1: *An instantiation of our reference architecture in terms of SCA Components*

Composite files

An SCA implementation can be specified either by *one* composite or by *as many* as components are in the application. For this implementation, each component of the reference architecture will be described by its own composite file to fulfill the extensible capability and distribution goals that were defined for this reference architecture.

The elements in a composite file for this implementation are:

- **component tag**: Defines the component name and other properties such as (i) Java source class implementation, (ii) Java source interface services definition, and (iii) the name of the references
- **service tag**: Defines for a component service the (i) communication protocol, (ii) machine where is located, (iii) the port in which the component service will be answering, and (iv) the name of the service.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="SensorRMIService"
targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">
diaponent name="SensorServiceComponent">

<implementation.java class="sensor.lib.SensorImpl" />
<service name="SensorService">
<interface.java interface="sensor.api.SensorService" />
</service>
<reference name="monitorService" />
</component>

<service name="SensorService" promote="SensorServiceComponent/SensorService">
<frascati:binding.rmi host="localhost" port="1102" serviceName="Sensor" />
</service>

<reference name="monitorService"
promote="SensorServiceComponent/monitorService">
<frascati:binding.rmi host="localhost" port="1100" serviceName="Monitor" />
</reference>

</composite>
```

Figure 4.2: *Composite File example for the Monitor Component*

- **reference tag**: Defines for a component reference the (i) communication protocol, (ii) machine where it will be found, (iii) the port in which the component service will be found, and (iv) the name of the service.

Figure 4.2 is an example of a composite file for this implementation, in this case for the Sensor component that will be deployed using an Open Source middleware platform for SCA components named FraSCAti implemented by the OW2 Consortium[2]. In this implementation both network interfaces are *localhost* because the implementation is running on the same machine, but if it is required a distribution of the components on different machines, the network interface in the *reference tag* of the composite files must be replaced with the IP Address were the required component is running.

As specified by the <*implementation.java*> tag, this component is implemented by the Java class *sensor.lib.SensorImpl* and its service is provided by the Java interface *sensor.api.SensorService*. This component provides a **service** named *SensorService* using RMI as a communication protocol, and is located in the *localhost* network interface through the port 1102 by the name *Sensor*. Also,

[2]FraSCAti official web site: http://wiki.ow2.org/FraSCAti/Wiki.jsp?page=FraSCAti

this component requires a service from the Monitor component, thus it defines a **reference** named *monitorService* with RMI communication protocol and will search it in the *localhost* network interface through the port 1100 by the name *Monitor*. According to this composite file both components will be running in the same machine. The complete set of composite files are presented in Appendix C.

4.1.2 Java Implementation

Figure 4.3 describes the general implementation of the components, which is based on two packages (i) *component.api* where the Java service Interface is defined with the services that the component provides and an Abstract class is defined with the basic implementation for the methods, services and references from other components in order to guarantee an extensible capability for our reference architecture, and (ii) *component.lib* where the Java implementation classes are defined with the specific behavior required by this case of study.

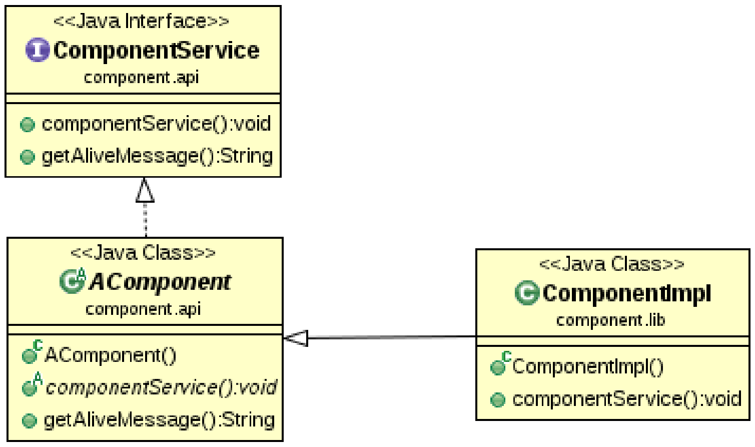


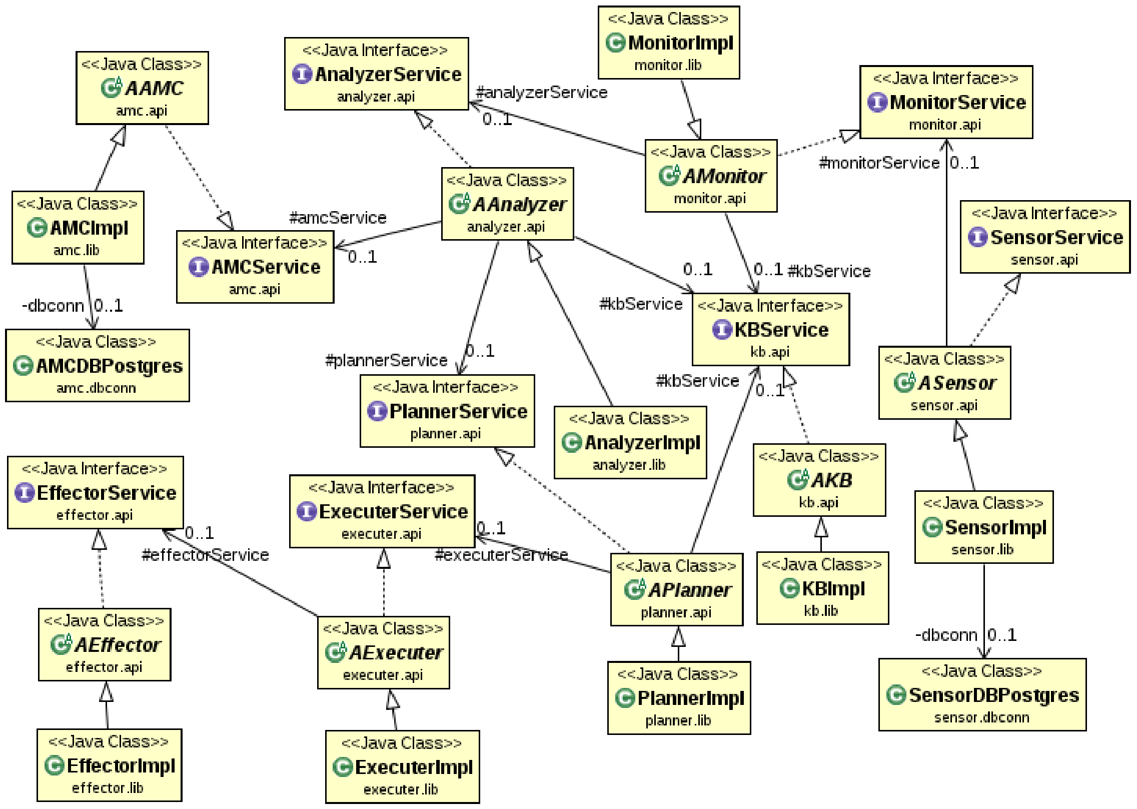
Figure 4.3: *The general Java implementation classes and packages for the MAPE-K components*

MAPE-K

Figure 4.4 describes the Java class diagram for each SCA components in the reference architecture and the relationship among these classes. For each component we can see the general structure applied and it is important to notice that the *Abstract* class is connected to the *other services interface*, this shows the relationships among the components. Context-Entities and Target System components described as separated elements in the reference architecture are both related to the RVCS software in this implementation, and the information about changes is stored in a RVCS server database for the use of the monitor probes. This is the reason why the Sensor Implementation *sensor.lib.SensorImpl* has an object form a class *sensor.dbconn.SensorDBPostgres* with the implementation for the database connection to gather the context changes information. Another important class is the *amc.lib.AMCImpl* with an *amc.dbconn.AMCDBPostgres* object that will search the information in the *mapek database*[3] to build the SLOs required to perform the adaptation analysis. Each abstract class defines the behavior of the components and relationships

[3]See Appendix A.2 to further information of the technical installation and configuration of this database

with other components, but for this case of study some specialization was required and the classes *sensor.lib.SensorImpl, monitor.lib.MonitorImpl, analyzer.lib.AnalyzerImpl, amc.lib.AMCImpl* and *kb.lib.KBImpl* has some particular implementation particular to this case of study and must be modified if other case of study is intended. However, as mentioned before, this implementation is meant to be extensible and the interfaces and abstract classes are sufficient to provide this capability.



<<Java Class>> AAMC amc.api
<<Java Interface>> AnalyzerService analyzer.api
<<Java Class>> MonitorImpl monitor.lib
<<Java Interface>> MonitorService monitor.api
<<Java Class>> AMonitor monitor.api
<<Java Class>> AMCImpl amc.lib
<<Java Interface>> AMCService amc.api
<<Java Class>> AAnalyzer analyzer.api
<<Java Interface>> SensorService sensor.api
<<Java Class>> AMCDBPostgres amc.dbconn
<<Java Interface>> KBService kb.api
<<Java Class>> ASensor sensor.api
<<Java Interface>> PlannerService planner.api
<<Java Class>> AnalyzerImpl analyzer.lib
<<Java Class>> AKB kb.api
<<Java Class>> SensorImpl sensor.lib
<<Java Interface>> EffectorService effector.api
<<Java Interface>> ExecuterService executer.api
<<Java Class>> APlanner planner.api
<<Java Class>> KBImpl kb.lib
<<Java Class>> AEffector effector.api
<<Java Class>> AExecuter executer.api
<<Java Class>> PlannerImpl planner.lib
<<Java Class>> SensorDBPostgres sensor.dbconn
<<Java Class>> EffectorImpl effector.lib
<<Java Class>> ExecuterImpl executer.lib
#analyzerService 0..1
#monitorService 0..1
#amcService 0..1
#kbService 0..1
-dbconn 0..1
#plannerService 0..1
#effectorService 0..1
#executerService 0..1

Figure 4.4: *MAPE-K Java Class Diagram (The complete version is presented in Appendix B.1 Fig.B.1)*

4.2 Application to the Case of Study

As mentioned in Section 3.1, the case of study for this project implies the interaction of two applications, one *end-user application* running in a mobile client, and a *server application*. This case of study is based on the running example presented by Tamura *et al.*. [24] which uses a simplified version of a Reliable Video-Conference System (RVCS) that we use to illustrate the requirements for dynamic reconfiguration.

4.2.1 The RVCS Scenario

The RVCS comprises two software applications in which self-adaptation takes place. Both will be evaluated separately in order to identify the context entities required to be monitored in each system and to design the appropriate mechanism to perform the self-adaptation.

Figure 3.1 in page 15 describes the interaction between the two software applications. A *videoconference mobile client* **registers** and **attends** videoconferences that are hosted in a *videoconference server*. Both applications establish the **Quality-of-Service (QoS) conditions** to be fulfilled by the videoconference. The self-adaptation mechanism for both applications must rely on the context changes, for example, a switch in the wireless connection of the client or an increase in the registration requests in the server. In any case, the RVCS must operate in the right conditions to fulfill the service agreements and perform the adaptation effectively.

4.2.2 Quality of Service Requirements

In order to maintain fulfilled the required QoS levels under the different context conditions faced by the target system, the Adaptation Mechanism is required to perform some specific activities. Table 3.1 shows these activities for each component *(as described in Chapter 3)* in both systems, and the context changes intended to be monitored. These activities correspond to the fulfillment of some quality attributes: (1) Confidentiality, (2) Availability and (3) Throughput. Tamura *et al.* defined the QoS contracts for these quality attributes [24]. In this case of study, we guarantee the **Throughput** quality attribute in the RVCS Server application in which the number of registration requests are computed as transactions per minute and the maximum limit that the server is capable to attend is delimited by the day of the month in the SLO Contract.

Table 4.1 summarizes for this contract the context events and the service level objectives be fulfilled (SLO). For this case a Context Event is the day of the month and the SLO is the maximum of transactions per minute allowed for the server. In the case in which this maximum is surpassed the adaptation mechanism must reconfigure the RVCS Server, otherwise the adaptation mechanism performs no operation.

Context Events	**SLO**
Day of the month	Transactions per minute
1-15	100
16-23	150
24-27	300
28-31	900

Table 4.1: *QoS contractual conditions and corresponding SLO for Throughput as presented by Tamura et al. [24]*

However, for this implementation it was defined a representation of the SLO Contracts as Java Objects. Figure 4.5 describes the Java implementation of the SLO Contracts and Monitoring policies. The reference model is the SLO contract defined by Tamura *et al.* [24]. According to this, Figure 4.6 describes the Object representation of the SLO Throughput contract and Appendix B.3 the java implementation lo load this information. As mentioned in Section 3.4.2 the AMC component is responsible for the delivery of this SLO so the Analyzer can analyze the information received form the Monitor by matching it against the SLO and finally decide whether adaptation must take place in the target system.

In this case of study, the AMC gathers the information from the *mapek database*[4] for the Throughput contract, in this case a single data table named **c3-predicate** to store the predicates

[4]This is a database created with the purpose of storing information of the contract

that are represented as *SLOPredicate* objects, in this table we have two columns: **observation**, to store the *Day of the month* and **value** to store the *Transaction per minute* following the information mentioned in Table 4.1.

The QoScontract name is a String resulted by adding the Quality Attribute *Throughput* and the code *C3* that corresponds to the Third Contract defined in Section 4.2.2. then, the QoSProperty is assigned by the constant THROUGHPUT defined by the QoSProperty class and the SLOObligation defines the type of context event for this contract, in this case defined by the constant TYPE_METHOD because this quality attribute will depend on the *Registration* method in the RVCS Server. This SLOObligation object has a relation with SLOPredicate (described before) and ContextCondition classes. The ContextCondition class represents what it is to be observed in the context to check this contract. For this implementation two observations are required: **Day of the month** and **Transactions per minute**, both are integer data types, and no mathematical operation is required.

As noticed in Figure 4.6, two extra classes are described: The MonitoringPolicy class that expresses the context entities or conditions that are meant to be observed, and the Observation definition during the monitoring. For this case of study, the MonitoringPolicy indicates that the type of Observation is TYPE_INVOKE which means that some service (in this case is the registration service) will be invoked in the context, and in a period of 60 seconds, the Monitor component must count these events, in this specific case, the amount of times the registration method is invoked. As a MonitoringPolicy is related to a SLO Contract, the monitoringContext object is the same contextCondition of the SLOObligation class, which means that the Monitor must be aware of information related to the Day of the Month and the Transactions per minute. These monitoring policies definitions are build by the Knowledge Base component as Java objects derived from the SLO contracts in the AMC component, and are delivered to the Monitor component.

This implementation uses a very simple Analyzing Policy based on a numeric evaluation: The Day of the month in the current execution is compared to one of the SLOPredicate observation (i.e. today is the 3rd and in the SLOPredicate observation array is the 3rd position). Then the Transactions per minuted in the current execution are compared to the value array same position of the day of the month (i.e. as today is the 3rd position, then the 3rd position of the value array is taken to comparison), if the evaluation is below or equal the the SLO is being fulfilled. If it is greater then the SLO is not fulfilled and an adaptation is required.

Notice that an Analyzing Policy might include a tolerance threshold, or an historical evaluation based on past behavior (For example what happened three days ago might affect today's decision), or even another non mathematical evaluation. This Analyzing Policy will depend on the type of information in the SLO Contract and the type information gathered by the context that will no necessarily be numbers, they could also be predicates encoded in Strings.

4.2.3 The Adaptation Mechanism for the RVCS

As mentioned previously, it is necessary that the adaptation mechanism must be separated from the system itself. Figure 4.7 describes the desired behavior for this self-adaptive software system. The RVCS provides two services, the first is a service called *getRegistrationActivity* with the registration information and consumed by the Adaptation Mechanism, and a second service called *feedAdaptationCommands* in which the Adaptation Mechanism can deliver all the adaptation execution plan so the RVCS can perform self-adaptation.

As mentioned in Section 4.1.2, for this implementation the RVCS server has an additional table

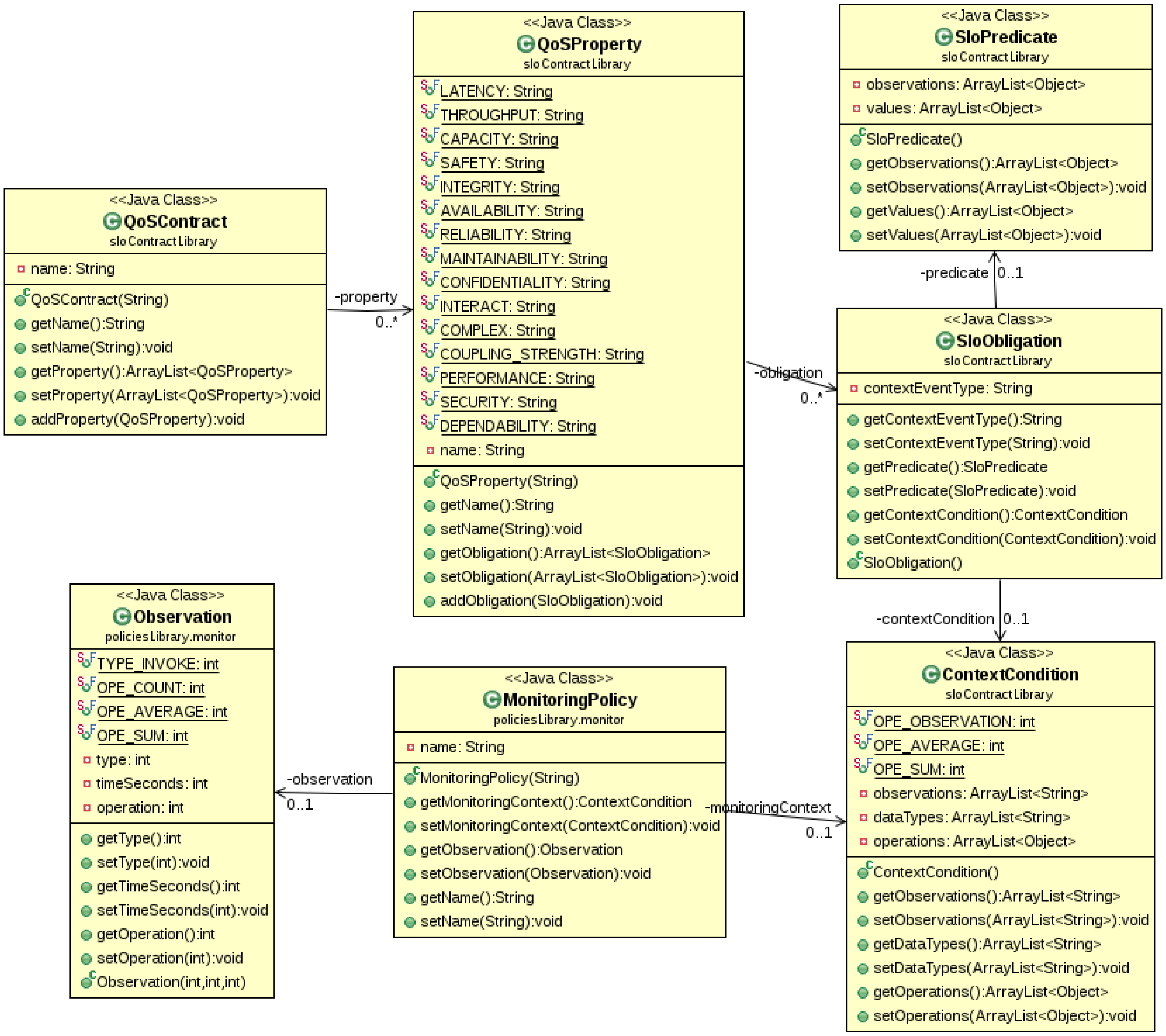


Figure 4.5: *Java Class Diagram for SLO Contracts and Monitoring Policies*

in the database in which is registered the registration information (other alternative could be the use of log files). The Sensor component implements a class to connect to the database and queries the information related to the last minute. As for the scope of the evaluation, this solution seems enough, however, other alternatives worth of future work, the Sensor component must specialize its touchpoints to be able to intrude the target application in the search of changes in the context.

Figure 4.8 describes the deployment of the applications configured for the evaluation. In this configuration three separated applications are executed: *(i)* The RVCS, *(ii)* the Adaptation Mechanism (AM), and *(iii)* the SoapUI Tester. The AM and the SoapUI Tester applications are both connected to the RVCS application. Notice that the AM is connected only to get the context changes information and it is not connected to deliver the adaptation instructions. The scope of

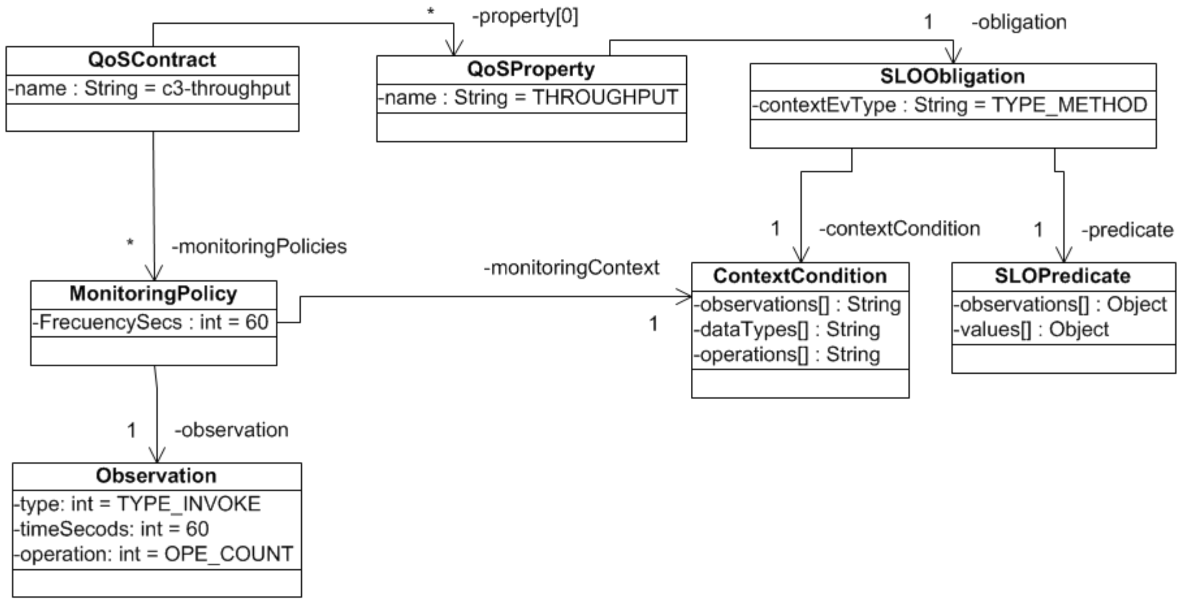


Figure 4.6: *Java Object Diagram for SLO Throughput Contract and Monitoring Policy*

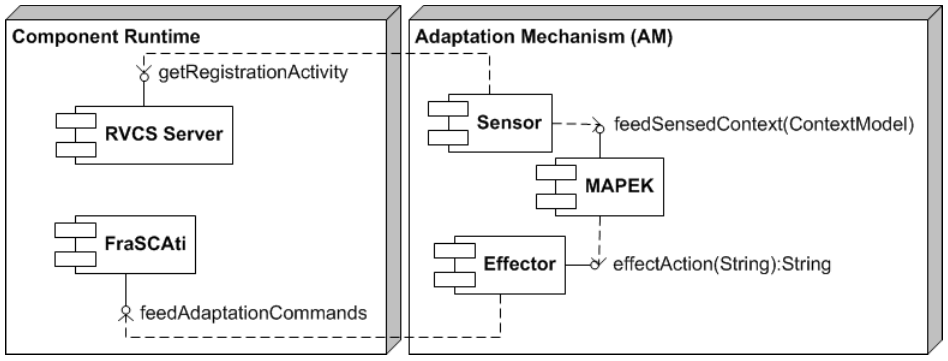


Figure 4.7: *RVCS and Adaptation Mechanism interaction*

this test does not provide the adaptation instructions but just a message indicating whether adaptation is required and some dummy instructions to guarantee the data flow through the Planner, Executer and Effector components. The adaptation Plan and Execution requires a more complex and specific work.

4.3 Test Cases

For this configuration the MAPE-K monitors the SLO Throughput Contract as described in Section 4.2. The general test case is described in Table 4.2 in which the **input** is the Day of the month *(d)* and the quantity of transactions received in the last minute *(x)*, the **evaluation** is the match between input with the current day range and the maximum transactions per minuted allowed, and the **output** depends on the result of the evaluation that can be either "Adaptation Required" or "No adaptation required". For this evaluation, Adaptation is required when x is greater than the maximum allowed.

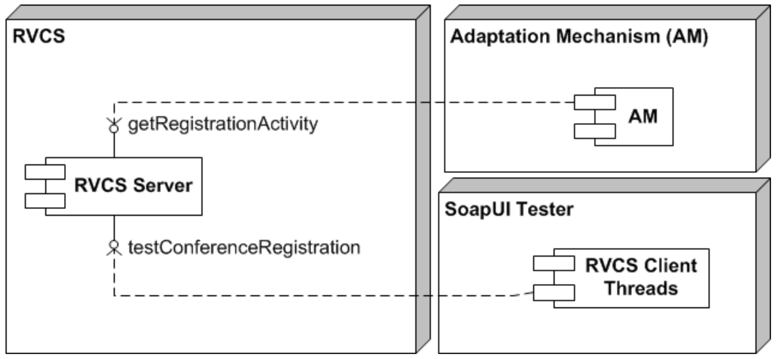


Figure 4.8: *RVCS and the MAPE-K deployment configuration set for the evaluation*

Input Context Information	**Evaluation** SLO condition	**Expected Output** MAPE-K Response
Day:d, Tx/min:x	$d \in$ Range. If $x <$Max Tx/min	No adaptation Required
Day:d, Tx/min:x	$d \in$ Range. If $x =$ Max Tx/min	No adaptation Required
Day:d, Tx/min:x	$d \in$ Range. If $x >$Max Tx/min	Adaptation Required

Table 4.2: *General conditions for the Test Cases for the MAPE-K evaluation corresponding the SLO for Throughput*

4.3.1 SoapUI Test Results

To generate the actual values for the concrete test cases we used SoapUI tool[5]. The RVCS service to test is "Registration to conference" in order to evaluate the SLO Throughput Contract.

In this test implementation, SoapUI stresses the RVCS system by creating a number of hypothetical attendants previously created in the RVCS system as *threadX* users where 'X' is a consecutive number corresponding to the thread generated by the SoapUI implementation. All these attendant clients will register to all the conferences available in the system. While the RVCS is running and attending these requests, the MAPE-K is also running and querying every minute for this type of requests registered in the database table. Table 4.3 describes the information and conditions during the test of the MAPE-K in this RVCS scenario. For this implementation RVCS applications and MAPE-K are running in the same machine. However, RVCS distribution does not affect the test because the context information is in the RVCS database.

Information	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**
Day of the month	15			16	
Conferences in the RVCS System	7				
Max Tx/min SLO	100			150	
Test Tx/min *(tx)*	tx = 0	tx <= 100	tx >100	100 <tx <= 150	tx >150
RVCS deployment	Single Machine				
MAPE-K deployment	Single Machine				

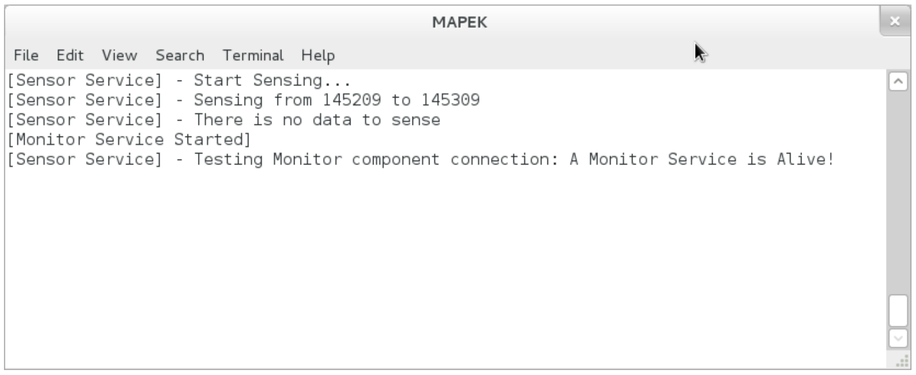
Table 4.3: *Contract Information for the Test Case scenario*

[5]Refer to Appendix A.5 for the complete implementation of the SOAP UI and the configuration files

Table 4.4 presents the results of the 5 tests performed over the described scenario in which the MAPE-K responses are expected showing the expected behavior of the MAPE-K's functionality. Figures 4.9, 4.10, 4.11, 4.12, 4.13 illustrate the output for some of the executed tests.

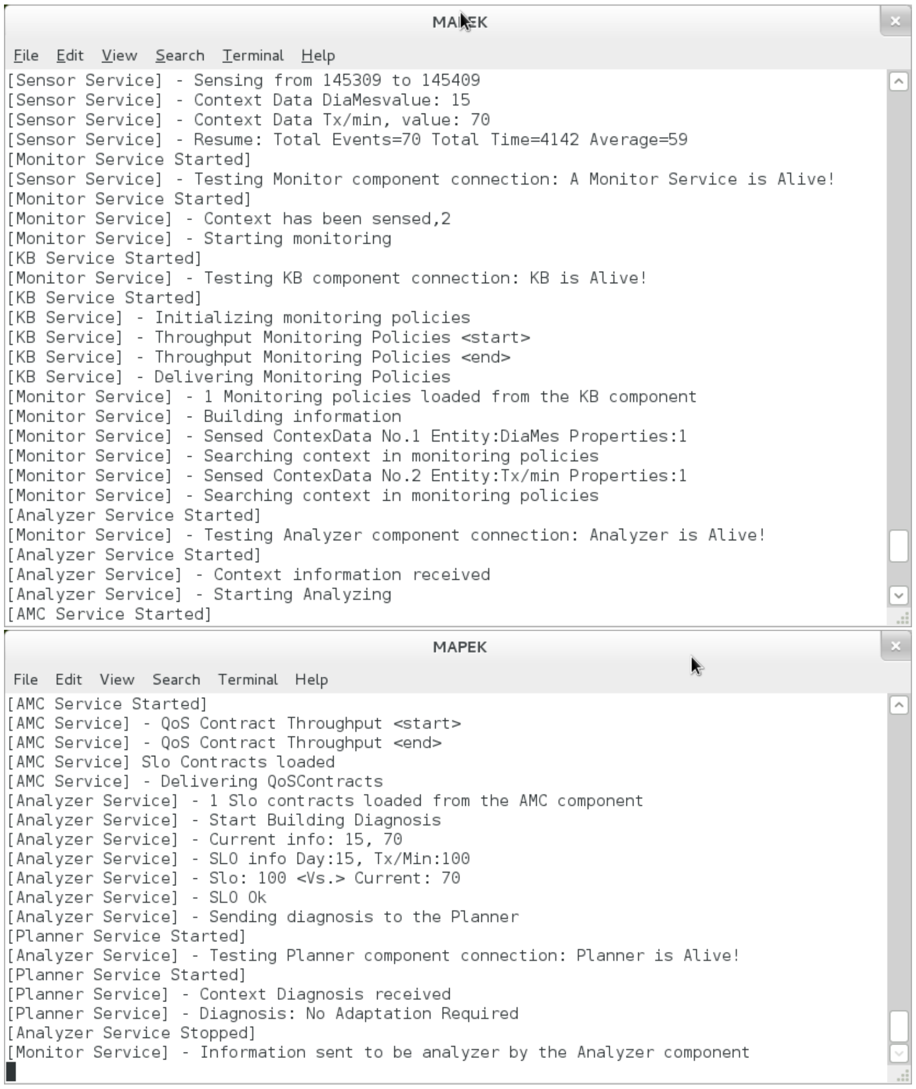
Test No.	**SoapUI Threads**	**RVCS Reg. Requests**	**SLO Information**	**Evaluation**	**MAPE-K Output**
1	0	0	Nothing to evaluate		
2	10	70	15 $\in$ Range:1-15 $\rightarrow$ Tx/min:100	Less	No adaptation
3	20	140	15 $\in$ Range:1-15 $\rightarrow$ Tx/min:100	Greater	Adaptation
4	20	140	16 $\in$ Range:16-23 $\rightarrow$ Tx/min:150	Less	No Adaptation
5	30	210	16 $\in$ Range:16-23 $\rightarrow$ Tx/min:150	Greater	Adaptation

Table 4.4: *Test results for the MAPE-K according the SLO for Throughput*



```
[Sensor Service] - Start Sensing...
[Sensor Service] - Sensing from 145209 to 145309
[Sensor Service] - There is no data to sense
[Monitor Service Started]
[Sensor Service] - Testing Monitor component connection: A Monitor Service is Alive!
```

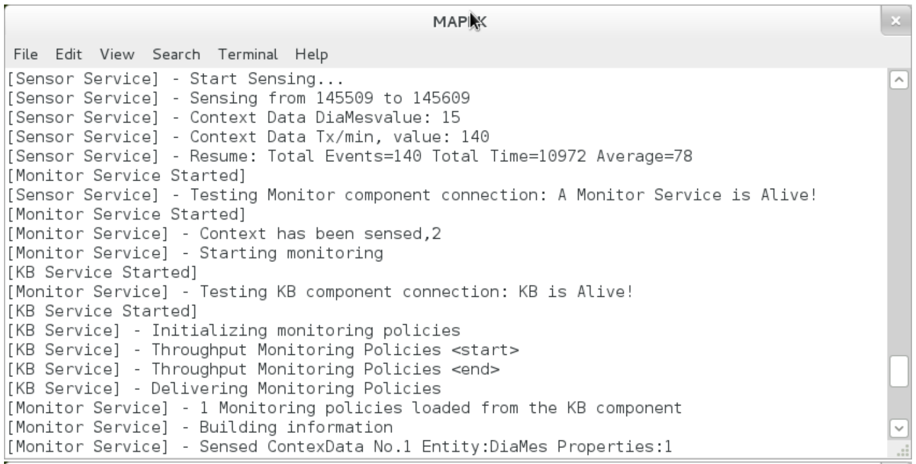
Figure 4.9: *Evaluation Test 1 Output example*



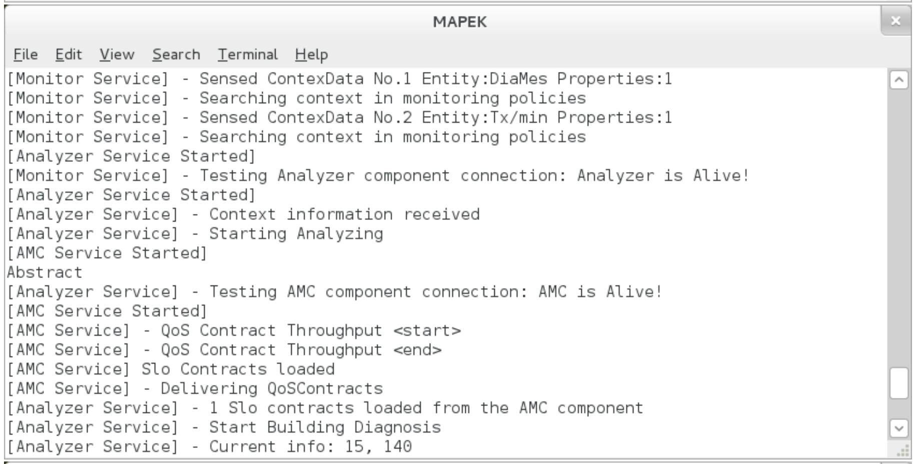
```
[Sensor Service] - Sensing from 145309 to 145409
[Sensor Service] - Context Data DiaMesvalue: 15
[Sensor Service] - Context Data Tx/min, value: 70
[Sensor Service] - Resume: Total Events=70 Total Time=4142 Average=59
[Monitor Service Started]
[Sensor Service] - Testing Monitor component connection: A Monitor Service is Alive!
[Monitor Service Started]
[Monitor Service] - Context has been sensed,2
[Monitor Service] - Starting monitoring
[KB Service Started]
[Monitor Service] - Testing KB component connection: KB is Alive!
[KB Service Started]
[KB Service] - Initializing monitoring policies
[KB Service] - Throughput Monitoring Policies <start>
[KB Service] - Throughput Monitoring Policies <end>
[KB Service] - Delivering Monitoring Policies
[Monitor Service] - 1 Monitoring policies loaded from the KB component
[Monitor Service] - Building information
[Monitor Service] - Sensed ContexData No.1 Entity:DiaMes Properties:1
[Monitor Service] - Searching context in monitoring policies
[Monitor Service] - Sensed ContexData No.2 Entity:Tx/min Properties:1
[Monitor Service] - Searching context in monitoring policies
[Analyzer Service Started]
[Monitor Service] - Testing Analyzer component connection: Analyzer is Alive!
[Analyzer Service Started]
[Analyzer Service] - Context information received
[Analyzer Service] - Starting Analyzing
[AMC Service Started]
```

```
[AMC Service Started]
[AMC Service] - QoS Contract Throughput <start>
[AMC Service] - QoS Contract Throughput <end>
[AMC Service] Slo Contracts loaded
[AMC Service] - Delivering QoSContracts
[Analyzer Service] - 1 Slo contracts loaded from the AMC component
[Analyzer Service] - Start Building Diagnosis
[Analyzer Service] - Current info: 15, 70
[Analyzer Service] - SLO info Day:15, Tx/Min:100
[Analyzer Service] - Slo: 100 <Vs.> Current: 70
[Analyzer Service] - SLO Ok
[Analyzer Service] - Sending diagnosis to the Planner
[Planner Service Started]
[Analyzer Service] - Testing Planner component connection: Planner is Alive!
[Planner Service Started]
[Planner Service] - Context Diagnosis received
[Planner Service] - Diagnosis: No Adaptation Required
[Analyzer Service Stopped]
[Monitor Service] - Information sent to be analyzer by the Analyzer component
```

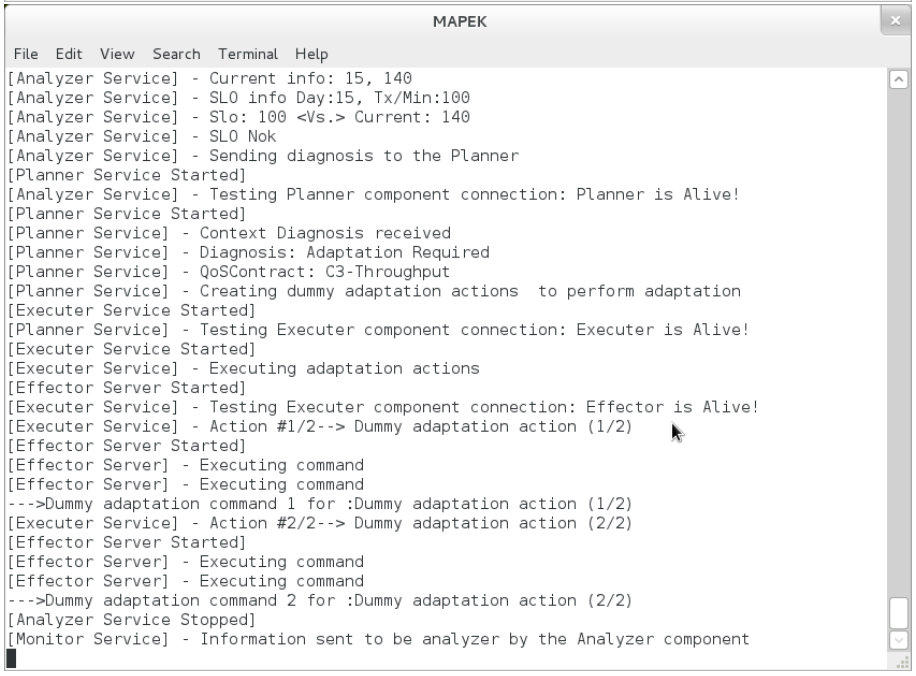
Figure 4.10: *Evaluation Test 2 Output example*



```
[Sensor Service] - Start Sensing...
[Sensor Service] - Sensing from 145509 to 145609
[Sensor Service] - Context Data DiaMesvalue: 15
[Sensor Service] - Context Data Tx/min, value: 140
[Sensor Service] - Resume: Total Events=140 Total Time=10972 Average=78
[Monitor Service Started]
[Sensor Service] - Testing Monitor component connection: A Monitor Service is Alive!
[Monitor Service Started]
[Monitor Service] - Context has been sensed,2
[Monitor Service] - Starting monitoring
[KB Service Started]
[Monitor Service] - Testing KB component connection: KB is Alive!
[KB Service Started]
[KB Service] - Initializing monitoring policies
[KB Service] - Throughput Monitoring Policies <start>
[KB Service] - Throughput Monitoring Policies <end>
[KB Service] - Delivering Monitoring Policies
[Monitor Service] - 1 Monitoring policies loaded from the KB component
[Monitor Service] - Building information
[Monitor Service] - Sensed ContexData No.1 Entity:DiaMes Properties:1
```

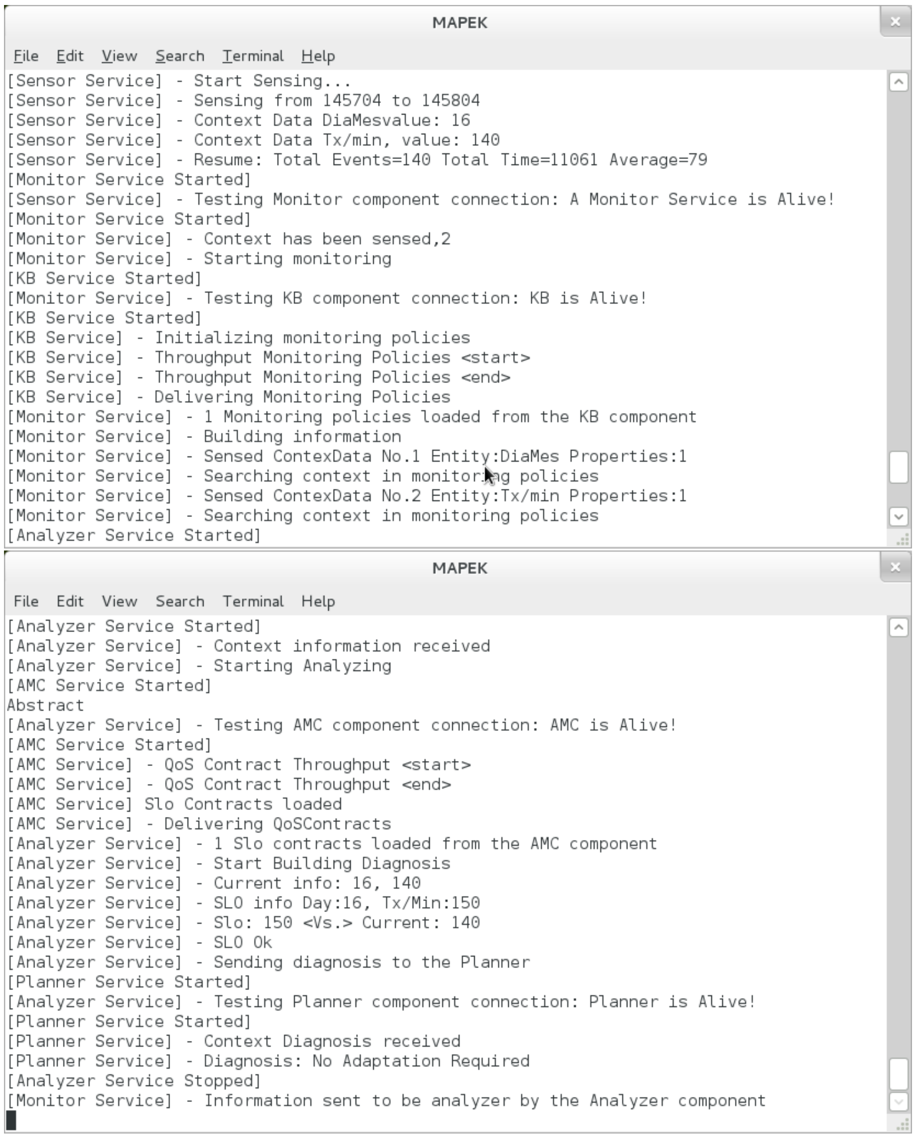


```
[Monitor Service] - Sensed ContexData No.1 Entity:DiaMes Properties:1
[Monitor Service] - Searching context in monitoring policies
[Monitor Service] - Sensed ContexData No.2 Entity:Tx/min Properties:1
[Monitor Service] - Searching context in monitoring policies
[Analyzer Service Started]
[Monitor Service] - Testing Analyzer component connection: Analyzer is Alive!
[Analyzer Service Started]
[Analyzer Service] - Context information received
[Analyzer Service] - Starting Analyzing
[AMC Service Started]
Abstract
[Analyzer Service] - Testing AMC component connection: AMC is Alive!
[AMC Service Started]
[AMC Service] - QoS Contract Throughput <start>
[AMC Service] - QoS Contract Throughput <end>
[AMC Service] Slo Contracts loaded
[AMC Service] - Delivering QoSContracts
[Analyzer Service] - 1 Slo contracts loaded from the AMC component
[Analyzer Service] - Start Building Diagnosis
[Analyzer Service] - Current info: 15, 140
```



```
[Analyzer Service] - Current info: 15, 140
[Analyzer Service] - SLO info Day:15, Tx/Min:100
[Analyzer Service] - Slo: 100 <Vs.> Current: 140
[Analyzer Service] - SLO Nok
[Analyzer Service] - Sending diagnosis to the Planner
[Planner Service Started]
[Analyzer Service] - Testing Planner component connection: Planner is Alive!
[Planner Service Started]
[Planner Service] - Context Diagnosis received
[Planner Service] - Diagnosis: Adaptation Required
[Planner Service] - QoSContract: C3-Throughput
[Planner Service] - Creating dummy adaptation actions  to perform adaptation
[Executer Service Started]
[Planner Service] - Testing Executer component connection: Executer is Alive!
[Executer Service Started]
[Executer Service] - Executing adaptation actions
[Effector Server Started]
[Executer Service] - Testing Executer component connection: Effector is Alive!
[Executer Service] - Action #1/2--> Dummy adaptation action (1/2)
[Effector Server Started]
[Effector Server] - Executing command
[Effector Server] - Executing command
--->Dummy adaptation command 1 for :Dummy adaptation action (1/2)
[Executer Service] - Action #2/2--> Dummy adaptation action (2/2)
[Effector Server Started]
[Effector Server] - Executing command
[Effector Server] - Executing command
--->Dummy adaptation command 2 for :Dummy adaptation action (2/2)
[Analyzer Service Stopped]
[Monitor Service] - Information sent to be analyzer by the Analyzer component
```

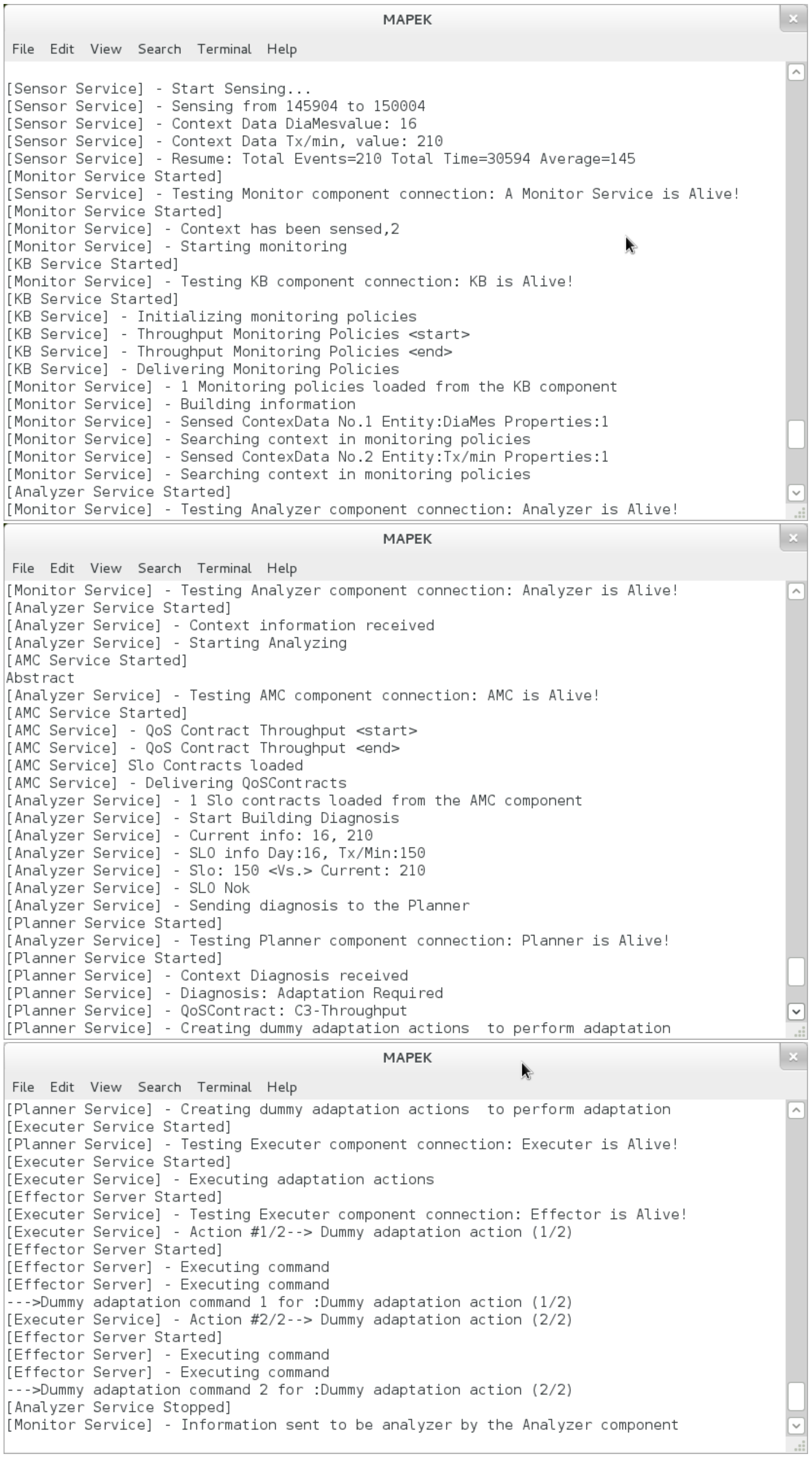
Figure 4.11: *Evaluation Test 3 Output example*


MAPEK
File Edit View Search Terminal Help
MAPEK
File Edit View Search Terminal Help

```
[Sensor Service] - Start Sensing...
[Sensor Service] - Sensing from 145704 to 145804
[Sensor Service] - Context Data DiaMesvalue: 16
[Sensor Service] - Context Data Tx/min, value: 140
[Sensor Service] - Resume: Total Events=140 Total Time=11061 Average=79
[Monitor Service Started]
[Sensor Service] - Testing Monitor component connection: A Monitor Service is Alive!
[Monitor Service Started]
[Monitor Service] - Context has been sensed,2
[Monitor Service] - Starting monitoring
[KB Service Started]
[Monitor Service] - Testing KB component connection: KB is Alive!
[KB Service Started]
[KB Service] - Initializing monitoring policies
[KB Service] - Throughput Monitoring Policies <start>
[KB Service] - Throughput Monitoring Policies <end>
[KB Service] - Delivering Monitoring Policies
[Monitor Service] - 1 Monitoring policies loaded from the KB component
[Monitor Service] - Building information
[Monitor Service] - Sensed ContexData No.1 Entity:DiaMes Properties:1
[Monitor Service] - Searching context in monitoring policies
[Monitor Service] - Sensed ContexData No.2 Entity:Tx/min Properties:1
[Monitor Service] - Searching context in monitoring policies
[Analyzer Service Started]
```

```
[Analyzer Service Started]
[Analyzer Service] - Context information received
[Analyzer Service] - Starting Analyzing
[AMC Service Started]
Abstract
[Analyzer Service] - Testing AMC component connection: AMC is Alive!
[AMC Service Started]
[AMC Service] - QoS Contract Throughput <start>
[AMC Service] - QoS Contract Throughput <end>
[AMC Service] Slo Contracts loaded
[AMC Service] - Delivering QoSContracts
[Analyzer Service] - 1 Slo contracts loaded from the AMC component
[Analyzer Service] - Start Building Diagnosis
[Analyzer Service] - Current info: 16, 140
[Analyzer Service] - SLO info Day:16, Tx/Min:150
[Analyzer Service] - Slo: 150 <Vs.> Current: 140
[Analyzer Service] - SLO Ok
[Analyzer Service] - Sending diagnosis to the Planner
[Planner Service Started]
[Analyzer Service] - Testing Planner component connection: Planner is Alive!
[Planner Service Started]
[Planner Service] - Context Diagnosis received
[Planner Service] - Diagnosis: No Adaptation Required
[Analyzer Service Stopped]
[Monitor Service] - Information sent to be analyzer by the Analyzer component
```

Figure 4.12: *Evaluation Test 4 Output example*


MAPEK
File Edit View Search Terminal Help
MAPEK
File Edit View Search Terminal Help
MAPEK
File Edit View Search Terminal Help

```
[Sensor Service] - Start Sensing...
[Sensor Service] - Sensing from 145904 to 150004
[Sensor Service] - Context Data DiaMesvalue: 16
[Sensor Service] - Context Data Tx/min, value: 210
[Sensor Service] - Resume: Total Events=210 Total Time=30594 Average=145
[Monitor Service Started]
[Sensor Service] - Testing Monitor component connection: A Monitor Service is Alive!
[Monitor Service Started]
[Monitor Service] - Context has been sensed,2
[Monitor Service] - Starting monitoring
[KB Service Started]
[Monitor Service] - Testing KB component connection: KB is Alive!
[KB Service Started]
[KB Service] - Initializing monitoring policies
[KB Service] - Throughput Monitoring Policies <start>
[KB Service] - Throughput Monitoring Policies <end>
[KB Service] - Delivering Monitoring Policies
[Monitor Service] - 1 Monitoring policies loaded from the KB component
[Monitor Service] - Building information
[Monitor Service] - Sensed ContexData No.1 Entity:DiaMes Properties:1
[Monitor Service] - Searching context in monitoring policies
[Monitor Service] - Sensed ContexData No.2 Entity:Tx/min Properties:1
[Monitor Service] - Searching context in monitoring policies
[Analyzer Service Started]
[Monitor Service] - Testing Analyzer component connection: Analyzer is Alive!
```

```
[Monitor Service] - Testing Analyzer component connection: Analyzer is Alive!
[Analyzer Service Started]
[Analyzer Service] - Context information received
[Analyzer Service] - Starting Analyzing
[AMC Service Started]
Abstract
[Analyzer Service] - Testing AMC component connection: AMC is Alive!
[AMC Service Started]
[AMC Service] - QoS Contract Throughput <start>
[AMC Service] - QoS Contract Throughput <end>
[AMC Service] Slo Contracts loaded
[AMC Service] - Delivering QoSContracts
[Analyzer Service] - 1 Slo contracts loaded from the AMC component
[Analyzer Service] - Start Building Diagnosis
[Analyzer Service] - Current info: 16, 210
[Analyzer Service] - SLO info Day:16, Tx/Min:150
[Analyzer Service] - Slo: 150 <Vs.> Current: 210
[Analyzer Service] - SLO Nok
[Analyzer Service] - Sending diagnosis to the Planner
[Planner Service Started]
[Analyzer Service] - Testing Planner component connection: Planner is Alive!
[Planner Service Started]
[Planner Service] - Context Diagnosis received
[Planner Service] - Diagnosis: Adaptation Required
[Planner Service] - QoSContract: C3-Throughput
[Planner Service] - Creating dummy adaptation actions  to perform adaptation
```

```
[Planner Service] - Creating dummy adaptation actions  to perform adaptation
[Executer Service Started]
[Planner Service] - Testing Executer component connection: Executer is Alive!
[Executer Service Started]
[Executer Service] - Executing adaptation actions
[Effector Server Started]
[Executer Service] - Testing Executer component connection: Effector is Alive!
[Executer Service] - Action #1/2--> Dummy adaptation action (1/2)
[Effector Server Started]
[Effector Server] - Executing command
[Effector Server] - Executing command
--->Dummy adaptation command 1 for :Dummy adaptation action (1/2)
[Executer Service] - Action #2/2--> Dummy adaptation action (2/2)
[Effector Server Started]
[Effector Server] - Executing command
[Effector Server] - Executing command
--->Dummy adaptation command 2 for :Dummy adaptation action (2/2)
[Analyzer Service Stopped]
[Monitor Service] - Information sent to be analyzer by the Analyzer component
```

Figure 4.13: *Evaluation Test 5 Output example*

Chapter 5

Conclusions and Future Work

One of the most important issues in self-adaptation is to create the capability in the system to reason about itself and its context. Control engineering uses feedback to measure and act upon the changes in the context that affect the behavior of the system. Moreover, *feedback loops* become important in the process the system must follow to measure a given property of the system and then evaluate if the desired goals are being reached [11]. Software engineers have made significant efforts to propose self-adaptive systems. However, these approaches are not general enough to satisfy the requirements that different systems, data types, and configurations can pose. Often, these proposals merge the adaptation mechanism with the target system making it very difficult to analyze the adaptation capability as an independent system, and in addition, feedback loops are not explicit in most of these proposals.

In this project we have developed a reference architecture that provides a general framework to implement adaptation mechanisms for software systems. This reference architecture allows to maintain a clear and complete separation of concerns between adaptation mechanisms and target systems, enabling the evaluation, maintenance and analysis of the adaptation process in a self-adaptive system. Moreover, each action in the adaptation process can be analyzed independently as part of a specific component in the reference architecture.

Based on components, this reference architecture also provides component reuse, the ability to be extensible and interoperable, which can help software engineers to specialize, extend and grow the functionalities of each component, according to the necessities of the specific target system. Moreover these components can be reused independently, as far as the target architecture provides the services required by them, which multiplies the reuse possibilities beyond the scenarios considered in this project.

As an SCA-compliant implementation, our reference architecture fulfills the goal of being extensible, which is one of the major goals of this project. Also, each SCA component of the reference architecture is defined in its own composite file in order to fulfill the distribution capability, which is the second major goal of this project.

Building a reference architecture for self-adaptive systems was one of the challenges proposed by Villegas et al. [30]. However, in this project we have identified an additional set of challenges worth of future work, such as the following:

- **Smoother touchpoints between the MAPE-K and the Context**
  As mentioned in Section 3.4.3 the *Sensor* and *Effector* components express the touchpoints defined by IBM's blueprint [12] connected to the Context and the Target System [29]. Our

implementation does not connect itself smoothly with the context. The desired behavior for this integration is that the Sensor would be embedded in the target system to analyze its behavior and then report to the MAPE-K about any change in the context. In the same way, the Target System exposes a service to receive the commands to perform adaptation, but a mutual agreement between the Effector and the Target System must be configured before. This is because the Effector, as an external component, must interpret the adaptation commands to be effected over the system.

- **Security between the Target System and the MAPE-K**
  As mentioned before, the Adaptation Mechanism is an external application that eventually, if adaptation is required, will perform operations over the Target System. Therefore, it is important the establishment of security policies and execution permissions between the Target System and the MAPE-K, so the Target System can be sure that this external component is reliable and can be trusted.

- **AMC development to introduce SLOs from humans into the MAPE-K System**
  In Section 4.2.2 we described our representation as Java objects for the SLO contracts defined by Tamura *et al.* [24]. However, translating SLO contracts from human language to objects requires a specific software implementation suitable for any user in charge of this task in the software system in a way that the system administrator can easily introduce or modify SLOs according the organization requirements that directly affect the adaptation process. Moreover, this SLO-definition task is highly important for the specification of the adaptation mechanism and its implementation must consider a verification and validation process, as well as a definition of *safe states* of SLO.

- **V&V after adaptation**
  As mentioned by Brun *et al.* [3] and Tamura *et al.* [26] the development of self-adaptive systems requires a quality mechanism to ensure that after adaptation the system is left in a valid state and the adaptation process finishes as expected. Otherwise the system should return to the last stable state known before adaptation. This validation and verification implies a challenge to this reference architecture because a control feedback loop must be necessary to evaluate the final state of the system according to a given quality criteria.

- **Planner and Executer extensions**
  As mentioned in Section 4.1.1 our implementation of the reference architecture does not include the Planner and Executer implementations because of the scope of this project. The Planner component tasks include the planning policies and techniques to generate the adaptation plan that eventually will modify the behavior of the target system. The Executer must transform the adaptation plan to commands that the Target System interprets to change itself according the new desired state. However, our implementation defines abstract Java classes for both components with the required behavior to complete the adaptation loop but do not effect any real change in the Target System, as implemented by Tamura [25].

Appendix A

Implementation Technical Guide

Follow the next section to deploy the implementation and test of this project.

A.1 Software requirements

1. Operative System : Fedora 16
2. Postgres database version 9
3. JBOSS 6.0.0.20100721-M4
4. Apache Maven 2.2.1
5. FraSCAti runtime 1.4
6. SOAP 2.3.1
7. Java Development Kit (jdk) v.16 update 25
8. RVCS v1.6
9. M2 repository for this project
10. MAPE-K v17052012
11. SOAP UI v-4.5.0

A.2 Installation Guide

A.2.1 Software installation

Use superuser privileges to follow these instructions:

- Install Postgres server with superuser *postgres* and password *postgresql* and the default port *5432*.
- Decompress SOAP file into /usr/local

- Decompress JBOSS file into /usr/local
- Decompress MAVEN file into /usr/local
- Configure the variables in the local user as in root user. Modify or add the lines in the *.bashrc* file in the home directory for both users.
- Decompress FraSCAti file into /usr/local
- Copy the Maven repository *.m2* into /root
- Decompress the RVCS file in /root
- Decompress the MAPE file in /root
- Install the SOAPUI application using the installation wizard

A.2.2 Environment Variables

Make sure you set local and global variables according your installation. Follow these as an example:

- JAVA_HOME in /usr/java/jdk1.6.0_25
- JBOSS_HOME in /usr/local/jboss-6.0.0.20100721-M4
- JMFHOME in /usr/local/jmf/JMF-2.1.1e
- FRASCATI_HOME /usr/local/frascati-runtime-1.4
- RVCS_HOME=/root/icesi/RVCS-SCA-v1.6

A.3 Setting Up the RVCS Scenario

For each execution step, open an individual terminal and log in as root user.

1. Create the **qoscaredb** database

    - Create a database named *qoscaredb* and an user *qoscare* with password *qoscare* and make it owner for the qoscaredb database.
    - With postgres privileges execute the scripts found in the *RVCS/qoscare scripts-db* folder, to load the qoscaredb in the following order: 1-drop-tables-generated.sql, 0-createall-generated.sql, 2-initial-data-generated.sql, 3-initial-data-generated.sql.

2. Execute **JBOSS** by executing *run.sh* located in */usr/local/jboss-6.0.0.20100721-M4/bin*
3. Execute **SCA Server** by executing *1-rvcs-scaserver.sh* located in */root/icesi/RVCS-SCA-v1.6/bin*
4. Execute **SCA Chat Server** by executing *./rvcs-scachatserver.sh* located in */root/icesi/RVCS-SCA-v1.6/bin*

5. Execute the **RVCS Client** executing in */root/icesi/RVCS-SCA-v1.6/src/scamodelclient* the line *mvn Prun*. As soon as the application is displayed log in with user and password *ikay* and in the dropdown list select *Attendant*.

6. Execute the **RVCS Testing service** executing in */root/icesi/RVCS-SCA-v1.6/src/scamodeltester* the line *mvn Prun*.

A.4 Setting Up the MAPE-K Implementation

1. Create the **mapek database**
   - Create a database named *mapek* and make user *qoscare* owner for the mapek database.
   - With postgres privileges execute the scripts found in the *MAPE-K170512* folder, to load the mapek in the following order: 1-create-mapekDB, 2-insert-mapekDB

2. Run the **MAPE-K implementation** by executing *MAPE-K/runMAPE.sh* and *runMAPE-K/run.sh*

A.5 Running the SOAPUI Test

Execute SOAPUI application and follow the next configuration:

1. Create a new project with name: **RVCS Tester**

2. Configure **Initial WSDL**: http://*RVCS-Testing-service-IP-Address*:9988/TesterServiceSCA?wsdl

3. Edit the file *Request 1* presented in Figure A.1 according with the following information:
   - **arg0** it is a number that corresponde the amount of threads the will be generated to request connection to the server
   - **arg1** it is an array of slots in which each of them is a RVCS client. For each slot the following information must be configured: *(i) currentCantRequest = 0*, *(ii) maxCantRequest = X* in which X represents the maximum quantity of connections that the client is going to allow before passing to the next slot or waiting list, *(iii) url =* is the URL address in which the test client wsdl is located (i.e., URL : http://*Cliente-IP-Address*:9876/ServiceExecutionSCA?wsdl)
   - **arg2** corresponds to the name of the service method for the test (i.e., testConferenceRegistration)

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:int="http://interfaces.tester.rvcs/" xmlns:bal="http://balancer.rvcs">
   <soapenv:Header/>
   <soapenv:Body>
      <int:testConferenceRegistration>
         <int:arg0>10</int:arg0>
```

```
            <!--Optional:-->
            <int:arg1>
               <!--Zero or more repetitions:-->
               <bal:Slot>
                  <!--Optional:-->
                  <bal:currentCantRequests>0</bal:currentCantRequests>
                  <!--Optional:-->
                  <bal:maxCantRequests>100</bal:maxCantRequests>
                  <!--Optional:-->
                  <bal:urlServiceTarget>
                  http://localhost:9876/ServiceExecutionSCA?wsdl
                  </bal:urlServiceTarget>
               </bal:Slot>
            </int:arg1>
            <!--Optional:-->
            <int:arg2>testConferenceRegistration</int:arg2>
         </int:testConferenceRegistration>
      </soapenv:Body>
   </soapenv:Envelope>
```

Figure A.1: *XML File configuration for SoapUI*

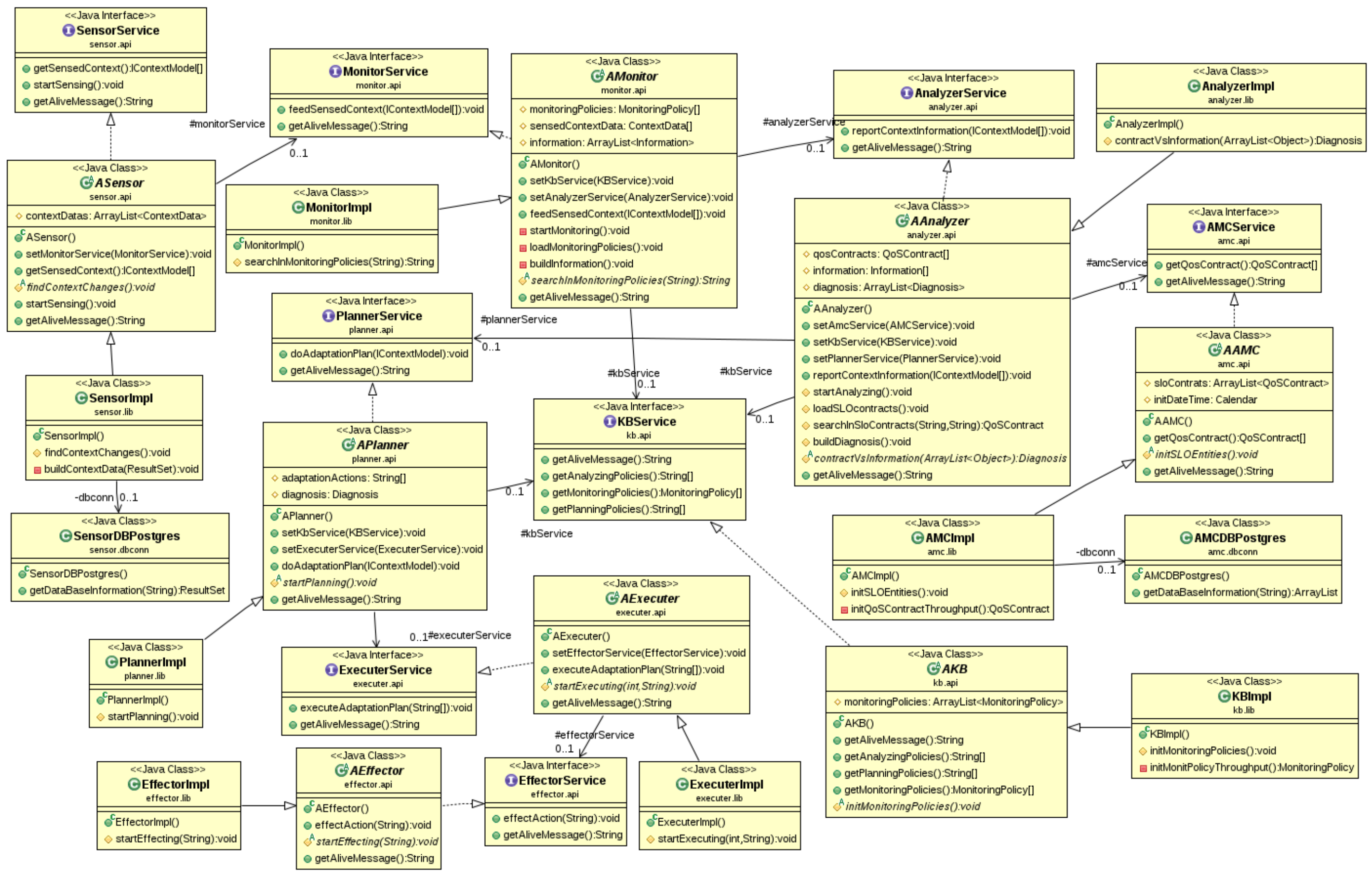
4. Execute SOAPUI test
5. Check the MAPE-K console for information about the execution of the test

Appendix B

Java Implementation Technical Guide

B.1 Reference Architecture Java Class Diagram

Figure B.1 shows the complete Java Class Diagram explained in Section 4.1.2.



<<Java Interface>> SensorService sensor.api
getSensedContext():IContextModel[]
startSensing():void
getAliveMessage():String
<<Java Interface>> MonitorService monitor.api
feedSensedContext(IContextModel[]):void
getAliveMessage():String
<<Java Class>> AMonitor monitor.api
monitoringPolicies: MonitoringPolicy[]
sensedContextData: ContextData[]
information: ArrayList<Information>
AMonitor()
setKbService(KBService):void
setAnalyzerService(AnalyzerService):void
feedSensedContext(IContextModel[]):void
startMonitoring():void
loadMonitoringPolicies():void
buildInformation():void
searchInMonitoringPolicies(String):String
getAliveMessage():String
<<Java Interface>> AnalyzerService analyzer.api
reportContextInformation(IContextModel[]):void
getAliveMessage():String
<<Java Class>> AnalyzerImpl analyzer.lib
AnalyzerImpl()
contractVsInformation(ArrayList<Object>):Diagnosis
<<Java Class>> ASensor sensor.api
contextDatas: ArrayList<ContextData>
ASensor()
setMonitorService(MonitorService):void
getSensedContext():IContextModel[]
findContextChanges():void
startSensing():void
getAliveMessage():String
<<Java Class>> MonitorImpl monitor.lib
MonitorImpl()
searchInMonitoringPolicies(String):String
<<Java Class>> AAnalyzer analyzer.api
qosContracts: QoSContract[]
information: Information[]
diagnosis: ArrayList<Diagnosis>
AAnalyzer()
setAmcService(AMCService):void
setKbService(KBService):void
setPlannerService(PlannerService):void
reportContextInformation(IContextModel[]):void
startAnalyzing():void
loadSLOcontracts():void
searchInSloContracts(String,String):QoSContract
buildDiagnosis():void
contractVsInformation(ArrayList<Object>):Diagnosis
getAliveMessage():String
<<Java Interface>> AMCService amc.api
getQosContract():QoSContract[]
getAliveMessage():String
<<Java Interface>> PlannerService planner.api
doAdaptationPlan(IContextModel):void
getAliveMessage():String
<<Java Class>> AAMC amc.api
sloContrats: ArrayList<QoSContract>
initDateTime: Calendar
AAMC()
getQosContract():QoSContract[]
initSLOEntities():void
getAliveMessage():String
<<Java Class>> SensorImpl sensor.lib
SensorImpl()
findContextChanges():void
buildContextData(ResultSet):void
<<Java Interface>> KBService kb.api
getAliveMessage():String
getAnalyzingPolicies():String[]
getMonitoringPolicies():MonitoringPolicy[]
getPlanningPolicies():String[]
<<Java Class>> APlanner planner.api
adaptationActions: String[]
diagnosis: Diagnosis
APlanner()
setKbService(KBService):void
setExecuterService(ExecuterService):void
doAdaptationPlan(IContextModel):void
startPlanning():void
getAliveMessage():String
<<Java Class>> SensorDBPostgres sensor.dbconn
SensorDBPostgres()
getDataBaseInformation(String):ResultSet
<<Java Class>> AMCImpl amc.lib
AMCImpl()
initSLOEntities():void
initQoSContractThroughput():QoSContract
<<Java Class>> AMCDBPostgres amc.dbconn
AMCDBPostgres()
getDataBaseInformation(String):ArrayList
<<Java Class>> AExecuter executer.api
AExecuter()
setEffectorService(EffectorService):void
executeAdaptationPlan(String[]):void
startExecuting(int,String):void
getAliveMessage():String
<<Java Class>> PlannerImpl planner.lib
PlannerImpl()
startPlanning():void
<<Java Interface>> ExecuterService executer.api
executeAdaptationPlan(String[]):void
getAliveMessage():String
<<Java Class>> AKB kb.api
monitoringPolicies: ArrayList<MonitoringPolicy>
AKB()
getAliveMessage():String
getAnalyzingPolicies():String[]
getPlanningPolicies():String[]
getMonitoringPolicies():MonitoringPolicy[]
initMonitoringPolicies():void
<<Java Class>> KBImpl kb.lib
KBImpl()
initMonitoringPolicies():void
initMonitPolicyThroughput():MonitoringPolicy
<<Java Class>> EffectorImpl effector.lib
EffectorImpl()
startEffecting(String):void
<<Java Class>> AEffector effector.api
AEffector()
effectAction(String):void
startEffecting(String):void
getAliveMessage():String
<<Java Interface>> EffectorService effector.api
effectAction(String):void
getAliveMessage():String
<<Java Class>> ExecuterImpl executer.lib
ExecuterImpl()
startExecuting(int,String):void
#monitorService
#analyzerService
#amcService
#plannerService
#kbService
#executerService
#effectorService
-dbconn
0..1

Figure B.1: *MAPE-K*

B.2 Context Library Java Class Diagram

Figure B.2 shows the complete Java Class Diagram explained in Section 3.3

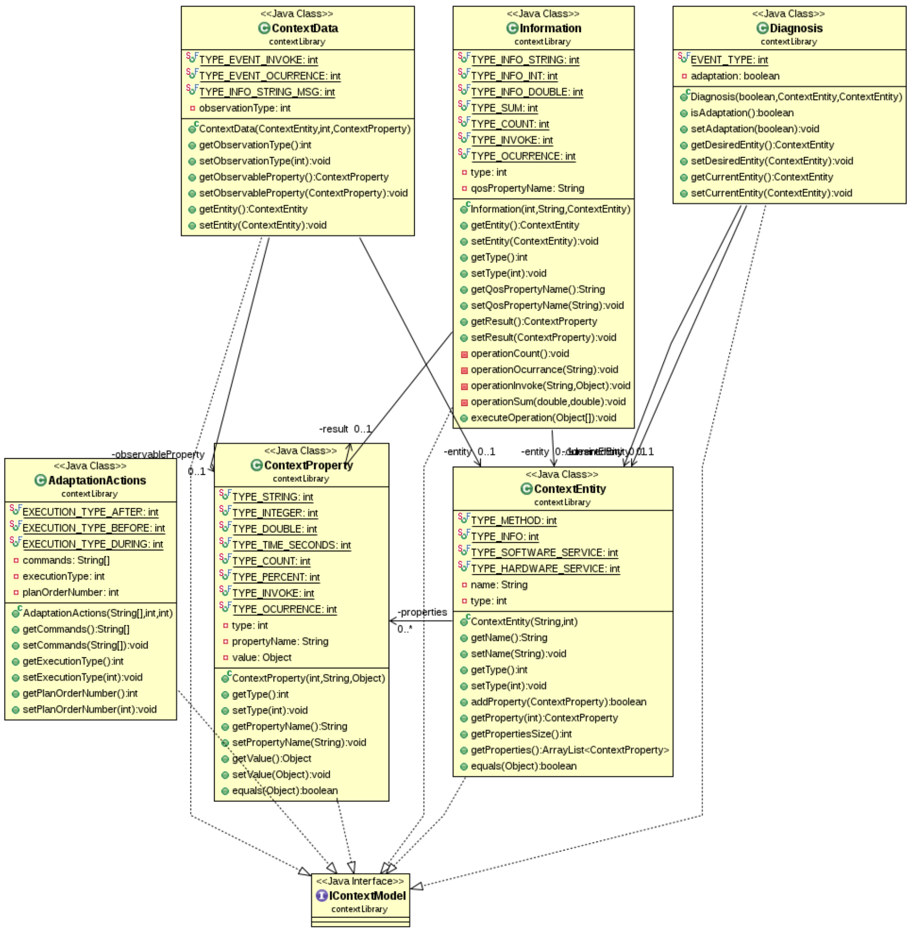


Figure B.2: *Context Library*

B.3 Java implementation to load the SLO Throughput Contract

The method *initQoSContractThroughput* defined in the *amc.lib.AMCImpl* class, loads from the mapek database the information to build the SLO Throughput Contract. As noticed,a QoSContract object is created an filled with the information, in this case, the day of the month and its corresponding maximum transactions per minute. Figure B.3 presents the java implementation for the AMCImpl class.

```
package amc.lib;

public class AMCImpl extends AAMC {
```

```
private AMCDBPostgres dbconn;

private QoSContract initQoSContractThroughput() {

System.out.println("[AMC Service] - QoS Contract Throughput <start>");
/*
 * The name of the contract is the sum of the qosContract ID and the QoS
 * Property to be assured C1-Confidentiality C2-Availability
 * C3-Throughput
 */
QoSContract contractThrougput = new QoSContract("C3-Throughput");

QoSProperty propertyP1 = new QoSProperty(QoSProperty.THROUGHPUT);
contractThrougput.addProperty(propertyP1);

SloObligation obligation = new SloObligation();
obligation.setContextEventType("Context Event Type Undefined");

SloPredicate predicate = new SloPredicate();
predicate.getObservations().add("DiaMes");
predicate.getObservations().add("Tx/min");

dbconn = new AMCDBPostgres();
ArrayList dbQuery = dbconn
.getDataBaseInformation("select * from \"c3predicate\"");

if (dbQuery.size() > 0) {

for (int i = 0; i < dbQuery.size(); i++) {
String[] val = (String[]) dbQuery.get(i);
int dia = Integer.parseInt(val[0]);
predicate.getValues().add(dia);
int tx = Integer.parseInt(val[1]);
predicate.getValues().add(tx);
}

obligation.setPredicate(predicate);

ContextCondition contextCondition = new ContextCondition();
contextCondition.getObservations().add("DiaMes");
contextCondition.getObservations().add("Tx/min");
contextCondition.getDataTypes().add("int");
contextCondition.getDataTypes().add("int");
contextCondition.getOperations().add(
ContextCondition.OPE_OBSERVATION);
contextCondition.getOperations().add(
ContextCondition.OPE_OBSERVATION);
obligation.setContextCondition(contextCondition);

propertyP1.addObligation(obligation);

System.out.println("[AMC Service] - QoS Contract Throughput <end>");

return contractThrougput;
} else {
System.out
.println("[AMC Service] - QoS Contract Throughput could not load"
+ "SLOContracts from data base");

return null;
}
}
}
```

Figure B.3: *Java Class Implementation for the AMC Abstract class*

B.4 Java implementation to load the Throughput Contract Monitoring Policies

The method *initMonitPolicyThroughput()* defined in the *kb.lib.KBImpl* class, loads from the SLO Throughput Contract the monitoring policies. Figure B.4 presents the Java implementation for the KBImpl class.

```
package kb.lib;

public class KBImpl extends AKB {

private MonitoringPolicy initMonitPolicyThroughput() {

System.out
.println("[KB Service] - Throughput Monitoring Policies <start>");

/*
 * The name of the policy is the sum of the qosContract ID and the QoS
 * Property to be assured C1-Confidentiality C2-Availability
 * C3-Throughput
 */
MonitoringPolicy policyThroughput = new MonitoringPolicy(
"C3-Throughput");

/*
 * ContextCondition that comes from the QoSContract C3 - Throughput at
 * KB
 */
ContextCondition contextCondition = new ContextCondition();
contextCondition.getObservations().add("DiaMes");
contextCondition.getObservations().add("Tx/min");
contextCondition.getDataTypes().add("int");
contextCondition.getDataTypes().add("int");
contextCondition.getOperations().add(ContextCondition.OPE_OBSERVATION);
contextCondition.getOperations().add(ContextCondition.OPE_OBSERVATION);

policyThroughput.setMonitoringContext(contextCondition);
policyThroughput.setObservation(new Observation(
Observation.TYPE_INVOKE, 60, Observation.OPE_COUNT));
System.out
.println("[KB Service] - Throughput Monitoring Policies <end>");

return policyThroughput;
}

}
```

Figure B.4: *Java Class Implementation for the KB Abstract class*

Appendix C

Composite Files

Figure C.1 presents the representation of our reference architecture as SCA components and describes the SCA Composite files. Note that each component has its own composite file and therefore are not explicit in the Figure. this Section describes each composite file in our implementation.

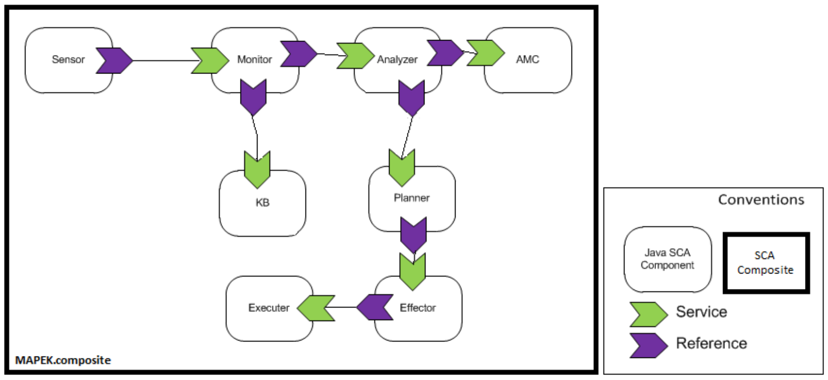


Figure C.1: *An instantiation of our reference architecture in terms of SCA Components and Composites*

C.1 MAPE-K.composite

The MAPE-K.composite file is the Composite that groups all the components in our reference architecture. In this file, we include all the other composites. Figure C.2 presents the Composite File for the reference architecture.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
 xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="MAPE-K-Loop">

  <component name="KB">
    <implementation.composite name="KBRMIService.composite"/>
  </component>

  <component name="AMC">
    <implementation.composite name="AMCRMIService.composite"/>
  </component>

  <component name="Effector">
    <implementation.composite name="EffectorRMIService.composite"/>
  </component>

  <component name="Executor">
    <implementation.composite name="ExecuterRMIService.composite"/>
  </component>

  <component name="Planner">
    <implementation.composite name="PlannerRMIService.composite"/>
  </component>

  <component name="Analyzer">
    <implementation.composite name="AnalyzerRMIService.composite"/>
  </component>

  <component name="Monitor">
    <implementation.composite name="MonitorRMIService.composite"/>
  </component>

  <component name="Sensor">
    <implementation.composite name="SensorRMIService.composite"/>
  </component>

</composite>
```

Figure C.2: *MAPE-K Composite file*

C.2 KBRMIService.composite

This Composite file described in Figure C.3 corresponds to the Knowledge Base (KB) component presented in Figure C.1. In this file the KB component exposes one service named **KB** through the network interface **localhost** and the port **1103**. This component does not require services from other components. The communication protocol described in this composite file is RMI *(Remote Method Invocation).*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="KBRMIService"
targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">

<component name="KBServiceComponent">

<implementation.java class="kb.lib.KBImpl"/>

<service name="KBService">
<interface.java interface="kb.api.KBService"/>
</service>

</component>

<service name="KBService" promote="KBServiceComponent/KBService">
<frascati:binding.rmi host="localhost" port="1103" serviceName="KB"/>
</service>

</composite>
```

Figure C.3: *Knowledge Base Composite file*

C.3 AMCRMIService.composite

This Composite File described in Figure C.4 corresponds to the Administrator Management Console (AMC) component presented in Figure C.1. In this file the AMC component exposes one service named **AMC** through the network interface **localhost** and the port **1199**. This component does not require services from other components. The communication protocol described in this composite file is RMI *(Remote Method Invocation)*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="AMCRMIService"
targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">

<component name="AMCServiceComponent">
```

```
<implementation.java class="amc.lib.AMCImpl"/>

<service name="AMCService">
<interface.java interface="amc.api.AMCService"/>
</service>

</component>

<service name="AMCService" promote="AMCServiceComponent/AMCService">
<frascati:binding.rmi host="localhost" port="1199" serviceName="AMC"/>
</service>

</composite>
```

Figure C.4: *Administrator Management Console Composite file*

C.4 EffectorRMIService.composite

This Composite file described in Figure C.5 corresponds to the Effector component presented in Figure C.1. In this file the Effector component exposes one service named **Effector** through the network interface **localhost** and the port **1106**. This component does not require services from other components. The communication protocol described in this composite file is RMI *(Remote Method Invocation)*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="EffectorRMIService"
targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">

<component name="EffectorServiceComponent">

<implementation.java class="effector.lib.EffectorImpl"/>

<service name="EffectorService">
<interface.java interface="effector.api.EffectorService"/>
</service>

</component>
```

```
<service name="EffectorService" promote="EffectorServiceComponent">
<frascati:binding.rmi host="localhost" port="1106" serviceName="Effector"/>
</service>

</composite>
```

Figure C.5: *Effector Composite file*

C.5 ExecuterRMIService.composite

This Composite file described in Figure C.6 corresponds to the Executer component presented in Figure C.1. In this file the Executer component exposes one service named **Executer** through the network interface **localhost** and the port **1105**. This component also requires a service named **Effector** through the network interface **localhost** and the port **1106**. The communication protocol described in this composite file is RMI *(Remote Method Invocation)*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="ExecuterRMIService"
targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">

<component name="ExecuterServiceComponent">

<implementation.java class="executer.lib.ExecuterImpl"/>

<service name="ExecuterService">
<interface.java interface="executer.api.ExecuterService"/>
</service>

<reference name="effectorService"/>

</component>

<service name="ExecuterService" promote="ExecuterServiceComponent">
<frascati:binding.rmi host="localhost" port="1105" serviceName="Executer"/>
</service>

<reference name="effector" promote="ExecuterServiceComponent/effectorService">
<frascati:binding.rmi host="localhost" port="1106" serviceName="Effector"/>
```

```
</reference>

</composite>
```

Figure C.6: *Executer Composite file*

C.6 PlannerRMIService.composite

This Composite file described in Figure C.7 corresponds to the Planner component presented in Figure C.1. In this file the Planner component exposes one service named **Planner** through the network interface **localhost** and the port **1104**. This component also requires two services, one is the service **Executer** through the network interface **localhost** and the port **1105** and the other is the service **KB** through the network interface **localhost** and the port **1103**. The communication protocol described in this composite file is RMI *(Remote Method Invocation)*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="PlannerRMIService"
targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">

<component name="PlannerServiceComponent">

<implementation.java class="planner.lib.PlannerImpl" />

<service name="PlannerService">
<interface.java interface="planner.api.PlannerService" />
</service>

<reference name="kbService" />
<reference name="executerService" />

</component>

<service name="PlannerService" promote="PlannerServiceComponent">
<frascati:binding.rmi host="localhost" port="1104"
serviceName="Planner" />
</service>

<reference name="kbService" promote="PlannerServiceComponent/kbService">
```

```
<frascati:binding.rmi host="localhost" port="1103"
serviceName="KB" />
</reference>

<reference name="executer" promote="PlannerServiceComponent/executerService">
<frascati:binding.rmi host="localhost" port="1105"
serviceName="Executer" />
</reference>

</composite>
```

Figure C.7: *Planner Composite file*

C.7 AnalyzerRMIService.composite

This Composite file described in Figure C.8 corresponds to the Analyzer component presented in Figure C.1. In this file the Analyzer component exposes one service named **Analyzer** through the network interface **localhost** and the port **1101**. This component also requires three services, one is the service **KB** through the network interface **localhost** and the port **1103**, other is the service **AMC** through the network interface **localhost** and the port **1199**, and the last one is the service **Planner** through the network interface **localhost** and the port **1104**. The communication protocol described in this composite file is RMI *(Remote Method Invocation)*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="AnalyzerRMIService"
targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">

<component name="AnalyzerServiceComponent">

<implementation.java class="analyzer.lib.AnalyzerImpl"/>

<service name="AnalyzerService">
<interface.java interface="analyzer.api.AnalyzerService"/>
</service>

<reference name="amcService"/>
<reference name="kbService"/>
<reference name="plannerService"/>
```

```
</component>

<service name="AnalyzerService" promote="AnalyzerServiceComponent">
<frascati:binding.rmi host="localhost" port="1101" serviceName="Analyzer"/>
</service>

<reference name="amcService" promote="AnalyzerServiceComponent/amcService">
<frascati:binding.rmi host="localhost" port="1199" serviceName="AMC"/>
</reference>

<reference name="kbService" promote="AnalyzerServiceComponent/kbService">
<frascati:binding.rmi host="localhost" port="1103" serviceName="KB"/>
</reference>

<reference name="plannerService"
promote="AnalyzerServiceComponent/plannerService">
<frascati:binding.rmi host="localhost" port="1104" serviceName="Planner"/>
</reference>

</composite>
```

Figure C.8: *Analyzer Composite file*

C.8 MonitorRMIService.composite

This Composite file described in Figure C.9 corresponds to the Monitor component presented in Figure C.1. In this file the Monitor component exposes one service named **Monitor** through the network interface **localhost** and the port **1100**. This component also requires two services, one is the service **KB** through the network interface **localhost** and the port **1103**, and the other is the service **Analyzer** through the network interface **localhost** and the port **1101**. The communication protocol described in this composite file is RMI *(Remote Method Invocation).*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="MonitorRMIService"
targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">

<component name="MonitorServiceComponent">
```

```
<implementation.java class="monitor.lib.MonitorImpl"/>

<service name="MonitorService">
<interface.java interface="monitor.api.MonitorService"/>
</service>

<reference name="kbService"/>
<reference name="analyzerService"/>

</component>

<service name="MonitorService" promote="MonitorServiceComponent/MonitorService">
<frascati:binding.rmi host="localhost" port="1100" serviceName="Monitor"/>
</service>

<reference name="kbService" promote="MonitorServiceComponent/kbService">
<frascati:binding.rmi host="localhost" port="1103" serviceName="KB"/>
</reference>

<reference name="analyzerService"
promote="MonitorServiceComponent/analyzerService">
<frascati:binding.rmi host="localhost" port="1101" serviceName="Analyzer"/>
</reference>

</composite>
```

Figure C.9: *Monitor Composite file*

C.9 SensorRMIService.composite

This Composite file described in Figure C.10 corresponds to the Sensor component presented in Figure C.1. In this file the Sensor component exposes one service named **Sensor** through the network interface **localhost** and the port **1102**. This component also requires a service named **Monitor** through the network interface **localhost** and the port **1100**. The communication protocol described in this composite file is RMI *(Remote Method Invocation)*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="SensorRMIService"
```

```
targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">

<component name="SensorServiceComponent">
<implementation.java class="sensor.lib.SensorImpl" />
<service name="SensorService">
<interface.java interface="sensor.api.SensorService" />
</service>
<reference name="monitorService" />
</component>

<service name="SensorService" promote="SensorServiceComponent/SensorService">
<frascati:binding.rmi host="localhost" port="1102"
serviceName="Sensor" />
</service>

<reference name="monitorService"
promote="SensorServiceComponent/monitorService">
<frascati:binding.rmi host="localhost" port="1100"
serviceName="Monitor" />

</reference>
</composite>
```

Figure C.10: *Sensor Composite file*

Bibliography

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture In Practice*. Addison-Wesley, 2003.

[2] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, and O. Hurley. Service Component Architecture, Assembly Model Specication. Specication Version 1.0. 2007.

[3] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, , and J. Whittle. Software engineering for self-adaptive systems: A research roadmap. *Software engineering for self-adaptive systems*, LNCS 5525:1 – 26, 2009.

[4] Microsoft Corp. Introducing Windows Communication Foundation in .NET Framework 4. http://msdn.microsoft.com/library/ee958158.aspx (Last visited: Jan/2012).

[5] R. Dawson, R. Desmarais, H.M. Kienle, and H.A Müller. Monitoring in Adaptive Systems Using Reflection. *ACM/IEEE*, SEAMS 2008:81 – 88, 2008.

[6] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems journal*, 42(1):5–18, 2003.

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

[8] David Garlan and Mary Shaw. *An Introduction to software architecture*. School of computer science. Carnegie Mellon University, 1994.

[9] H. Giese, Y. Brun, J. D. M. Serugendo, C. Gacek, H. Kienle, H. Müller, M. Pezzè, , and M. Shaw. Engineering self-adaptive systems through feedback loops. *Springer-Verlag*, LNCS 5525:47 – 69, 2009.

[10] George Heineman and William Councill. *Component-based software engineering*. Addison-wesley, 2007.

[11] J. L. Hellerstein, Y. Diao, S. Parekh, , and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley and Sons, 2004.

[12] IBM. An Architectural blueprint for autonomic computing. *Autonomic computing*, 2005.

[13] Oracle Inc. Java EE at a Glance. http://www.oracle.com/technetwork/java/javaee/overview/index.html (Last visited: Jan/2012).

[14] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer*, 36 No.1:41 – 50, 2003.

[15] J Kramer and J Magee. Self-managed Systems: an Architectural Challenge. *IEEE Computer*, pages 259 – 268, 2007.

[16] Philip K McKinley, Seyed Masoud Sadjadi, and Betty H.C. Cheng. Composing Adaptive Software. *IEEE Computer*, July 2004.

[17] H. Müller, M. Pezzè, , and M. Shaw. Visibility of control in adaptive systems. *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, ULSSIS 2008:23 – 26, 2008.

[18] H . A. Müller, H. M. Kienle, and U. Stege. Autonomic computing: Now you see it, now you dontdesign and evolution of autonomic software systems. *Lecture Notes in Computer Science*, 5413:32 – 54, 2009.

[19] OASIS Organization. Service Component Architecture (SCA) Specification. http://www.oasis-opencsa.org/sca (Last visited: Jan/2012).

[20] Carlos Parra, Xavier Blanc, and Laurence Duchien. Context Awareness for Dynamic Service-Oriented Product Lines. In John McGregor and Dirk Muthig, editors, *13th International Software Product Line Conference*, volume 1, pages 131–140, San Francisco, États-Unis, August 2009. Acceptance rate: 30/83 (36%). Rank (CORE) : B.

[21] Bogdan Solomon, Dan Ionescu, Marin Litoiu, and Mircea Mihaescu. A real-time adaptive control of autonomic computing environments. *CASCON*, pages 124 – 136, 2007.

[22] Thomas Stahl and Markus Vlter. *Model-Driven Software Development.* Jhon Wiley and Sons, Ltd, 2006.

[23] Clemens Szyperski. *Component Software. Beyond Object-Oriented Programming.* Addison-wesley, 2007.

[24] G. Tamura, C. Demarey, R. Casallas, and L. Duchien. QoS-CARE: A Framework for self-reliable QoS Contract Preservation through Self-Reconfiguration. *Preprint submitted to JSS Special Issue State of the Art in Self-Adaptive Systems*, 2012.

[25] Gabriel Tamura. *QoS-CARE: A Reliable System for Preserving QoS Contracts through Dynamic Reconfiguration.* Phd thesis, University of Lille 1 - Science and Technology and Universidad de Los Andes, May 2012.

[26] Gabriel Tamura, Rubby Casallas, Anthony Cleve, and Laurence Duchien. QoS contract-aware reconfiguration of component architectures using e-graphs. In *Proceedings of the 7th international conference on Formal Aspects of Component Software*, FACS'10, pages 34–52, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] Vijay Tewari and Milan Milenkovic. Standards for Autonomic Computing. *Intel technology journal*, 10 No.4:275 – 284, 2006.

[28] D. Truex, R. Baskerville, and H Klein. Growing Systems in Emergent Organizations. *Communications of the ACM*, 42 No. 8:117 – 123, 1999.

[29] N.M. Villegas and H.A Müller. Context-Driven Adaptive Monitoring for Supporting SOA Governance. *Proceedings of the 4th International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA 2010)*, Carnegie Mellon University Software Engineering Institute, 2010.

[30] Norha M. Villegas, Gabriel Tamura, Hausi A. Müller, Laurence Duchien, and Rubby Casallas. DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems 2*, volume 7475 of *LNCS*, pages 282 – 310. Springer, 2012.