A Three-Step Approach for Building Correct-by-Design Autonomic Service-Oriented Architectures

Emna Mezghani, Riadh Ben Halima, Ismael Bouassida Rodriguez, Khalil Drira

A Three-Step Approach for Building Correct-by-Design Autonomic Service-Oriented Architectures

Emna Mezghani[1,2,3], Riadh Ben Halima[1,2,3], Ismael Bouassida Rodriguez[1,2,3] and Khalil Drira[1,2]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France

[2] Univ de Toulouse, LAAS, F-31400 Toulouse, France

[3] University of Sfax, ReDCAD, B.P.W, 3038 Sfax, Tunisia

{emezghan,rbenhali,bouassida,drira}@laas.fr

Abstract: Autonomic systems are known by their ability to manage and reconfigure themselves in reaction to context changes without human intervention. The manual design and management of such complex systems is an error-prone task where both functional and non-functional requirements can be disturbed. In this paper, we provide a correct-by-design approach that allows a given abstract architectural description to be refined into autonomic architecture models that are close to implementations. The challenge is to get a system architecture that includes the necessary components for monitoring the non-functional parameters (e.g. quality of service) and reacting to any degradation by performing runtime reconfigurations. For solving such a problem, we provide an automated approach where an architecture is modelled as a conceptual graph with different levels of abstractions. Nodes represent software components or services or connectors and vertices represent communication or interaction links. To endow a given architecture with such properties, we define graph transformation rules to formally refine a given abstract representation into a specific model allowing the easy implementation of the autonomic schema. Such a refined schema includes the autonomic control loop components namely Monitoring, Analysis, Planning, and Execution (MAPE). We apply our approach to the "Campus-Wide Smart Metering" use-case providing a service-oriented style connecting Machine-to-Machine devices.

Key Words: Architecture, Autonomic computing, Correct by design, Dynamic reconfiguration.

1 Introduction

Service-Oriented Architectures (SOA) represent an abstract style to implement user requirements in software systems and their applications. These systems are built on top of loosely-coupled services which are accessible to a large community of clients. Still, the Service-Oriented Architecture abstraction contributes, but is not sufficient to manage the dynamic reconfiguration properties as required for adaptation to context changes and to provide the best Quality of Service (QoS).

The design of such systems is too complex. It is hard to solve systems entities dependencies when integrating new entities to the existing ones. It requires different refinement steps. The software architect has, first, to elaborate an initial abstract architecture description from the user requirements. Such

a model can be provided as a UML component diagram that includes business components and the related connections. Then, she/he adds incrementally both additional functional and non-functional requirements during the design process until the most refined description that can be associated with the real system design for the implementation. The Model Driven Architecture (MDA) [Miller and Mukerji, 2003] presents a modelling approach that starts from a high level model, and which refines this model in order to get the final model ready to be mapped to a given specific platform.

Few design approaches take into consideration the dynamic evolution of the non-functional requirements to reconfigure the system. Enabling that for SOA requires extending applications with the autonomic computing control loop that refers to monitor QoS parameters, to analyze them, to plan enforce, at runtime, reconfiguration actions [Riadh Ben-Halima, 2008].

However, it is tedious to incorporate the control loop components in order to support dynamic reconfiguration in SOA-compliant applications since the handmade refinement approaches are usually error-prone. Several research activities [Cristian Ruz and Sauvan, 2011], [Maurel et al., 2010], [Sheng et al., 2010], [Gauvrit et al., 2010] handle the dynamic reconfiguration and show different ways to adapt systems to the requirements. The variety of the solutions in the literature underlines the need to provide rule-based approaches and tools for software architects to assist them in building such systems.

In this paper, we define an automated approach that allows architects to generate correct-by-design refined representations of autonomic SOA-compliant for communicating systems and their applications. Starting from an abstract architectural model of functional and non-functional requirements, we define graph grammars [Chomsky, 1956a] transformation rules that govern the iterative refining process and incorporate the autonomic features. The refinement inside each step is based on graph transformation implemented using the Graph Matching and Transformation Engine (GMTE)[1].

The rest of this paper is organized as follows. In Section 2, we present different existing work in the field of transforming and refining systems. In Section 3, we detail our approach and describe the different steps. In Section 4, we give our motivating example and describe refinements performed to get autonomic Service-Oriented Architectures. And, finally, Section 5 concludes the paper and gives an overview of our future work.

2 Related Work

Starting from user requirements (the abstract view of the system) until reaching application implementation is the purpose of many researchers that adopt the

[1] The GMTE is available at `http://homepages.laas.fr/khalil/GMTE`

iterative refinement of architectures. Some of them, adapt MDA to tackle the separation of concerns at design time.

2.1 MDA Driven Approaches

Authors of [Cao et al., 2008] propose an MDA approach which focuses on transforming the CIM (Computational Independent Model) to the PIM (Platform Independent Model). This approach starts by modelling SOCIM (Service-Oriented CIM) requirements using the Service-Oriented Feature Model then they refine this model using several refinement rules and patterns in order to get the SOPIM (Service-Oriented PIM).

Research activities of [Rafe et al., 2009] and [Pena et al., 2006b] concentrate on the transformation from the PIM to the PSM (Platform Specific Model). Authors of [Rafe et al., 2009] present an MDA approach based on Story Driven Modelling and Aspect Oriented Programming for building SOA-compliant applications. They start from the PIM based on a UML standard profile to get the executable code. This approach is composed of three steps: transforming the PIM into PSM (SOA profile), then into a middleware independent code achieved via the concept of aspect and finally transforming this middleware into an executable code. Authors of [Pena et al., 2006b] present an MDA approach for applying policies to autonomic systems. This approach is based on a set of UML models and uses an Agent-Oriented methodology called MaCMAS [Pena et al., 2006a] to specify autonomous features. The authors proposed to use three PIM models and a PSM model. These models are generated by applying a set of transformations to the first PIM model (M-RAAF, Reusable Autonomous & Autonomic Features Model). Therefore, adding new policies is applied in the two first PIM model in order to add new features and the rest will be generated thanks to the transforming model. They validate their approach by the NASA ANTS case study.

In [Utomo, 2011], the author tackles the whole MDA levels and proposes an SOA-MDA method to implement an enterprise integration. This method provides a set of concepts that cover both business perspectives (business features and business requirements) and systems perspectives (functionalities for satisfying business requirements). The CIM describes the business perspectives using a Business Process Model, while the PIM and the PSM represent the systems perspectives. The PIM includes five phases: Abstract Layer Model, Use Case Model, Class Model, Sequence Model and Component Model. In the PSM level, where the SOA is integrated after modelling the PIM, they defined three phases related to SOA: WSDL Model, BPEL Model and Composite Model. They prove the success of their method through the e-Shop case study.

These research activities involve the MDA to model the different levels of abstraction of the application. They even detail each level into sub-levels like the

work of [Utomo, 2011] and [Pena et al., 2006b]. However, there are no details provided about the transformations. Except the work of [Pena et al., 2006b], they do not consider autonomic systems. Moreover, none of them discusses how it is possible to model, transform and automatically generate dynamically reconfigurable architectures.

2.2 Multi-Level Model-Driven Approaches

In this context, Sancho's [Sancho, 2009] and Rodriguez et al. [Rodriguez et al., 2010] present a multi-level model-driven reconfiguration approach for collaborative and ubiquitous systems. Their work focuses on processing transformation models based on ontology using graph transformation techniques for refining architectural configurations for multi-level architecture systems (application, collaboration and middleware levels).

The authors of [Baresi et al., 2004] propose a correct refinement of abstract architectural models into platform specific representations. Their approach aims at checking and refining the dynamic architectures. They used graph transformation and rule-based refinements to model architectural styles for different levels of platform abstraction and to automatically map them to specific platforms while preserving the semantic correctness. They validated their approach with an SOA-compliant case study namely "room reservation system".

However, the refinement of an abstract architecture to get a dynamic reconfigurable structurally equivalent architecture is a hard task for the architect and increases with the complexity of the systems. Therefore, automating the design of such systems is crucial. We propose a rule-based approach that takes advantages of the graph grammar strength in order to tackle the complexity of designing autonomic architectures. We handle non-functional requirements with a focus on QoS management for SOA-compliant applications.

3 A Rule-Based Transformation Approach

In this section, we present an overview of our three-step modelling approach. We focus on how to generate an autonomic SOA-compliant architecture for communicating systems and their applications. We give also details for each step.

3.1 The Approach

In software engineering, we notice that designing and developing autonomic applications are not an easy task. It needs an expert on autonomic computing in order to understand the requirements and then plug the autonomous components in the designed architectural model. The proposed approach aims at formalizing this expertise and automating the integration of autonomic components at design

time. Our main goal is to assist software architects with an efficient approach that automates the design of autonomic Service-Oriented Architectures.

3.1.1 An Overview

Starting from an abstract architectural model that describes the functional requirements, our approach allows incorporating autonomic components that manage the system monitoring and reconfiguration according to its non-functional requirements. Following the autonomic management architecture proposed by [Kephart and Chess, 2003], managing non-functional requirements needs four autonomic control loop components: Monitoring which observes QoS parameters, Analysis which detects and identifies QoS degradation, Planning which plans for reconfiguration actions, Execution which enforces them.

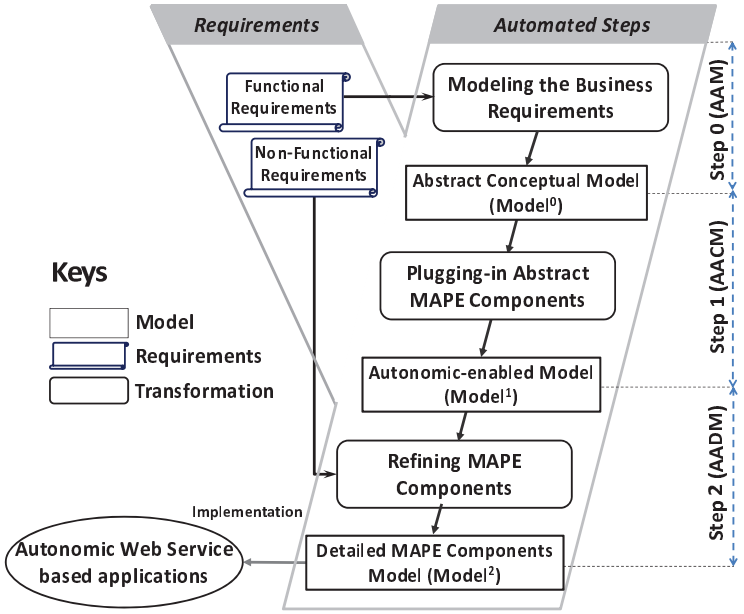

Figure 1: The three-step transformation approach

As shown in Figure 1, our approach includes two branches. The left-hand branch is dedicated to the requirements description. The right-hand branch is dedicated to the architecture description models and their specification and refinement steps. We distinguish three steps. *Step 0* is called Initial Abstract Architectural Model (AAM). In this step, the architect manually provides an abstract

description based on the functional requirements. It is represented as a UML component diagram without being dependent on UML notations. Other kinds of abstract descriptions can be specified during this step. *Step 1* is called Autonomic Architecture Component Model (AACM) in which we automatically plug in the four autonomic control loop components. Details of each autonomic control loop component are generated during *step 2* that we call Autonomic Architecture Deployment Model (AADM). It refines the content of each component regarding the non-functional requirements. As a result, we get an autonomic architectural model able to monitor its state and dynamically reconfigure itself at runtime if a QoS degradation occurs. The implementation of such dynamic architectural model as an SOA can lead to an autonomic Web Service based application.

3.1.2 The Transformation Process

The automation of *step 1* and *step 2* is based on graph transformation using graph grammars. So, each architectural model is automatically mapped to a graph, and then transformed using graph transformation rules. The graph transformation takes an important role because it can automate model transformation. At present, few graph transformation tools [Gei et al., 2006, Ermel et al., 1999] support multi-labelled graph inputs. Also, the main problem of these tools lies on the limited expressiveness of the used transformation rules. This kind of issue negatively reduces the range of application scenarios to be modelled and/or negatively increases the number of needed transformation rules. In this paper, we use the tool GMTE which handles directed and multi-labelled graphs. It allows performing graph matching and sequential transformation of graphs as well as rule applications as long as possible.

A generative grammar [Chomsky, 1956b] is defined, in general, as a classical grammar system $< AX; NT; T; P >$, where AX is the axiom, NT is the set of the non-terminal nodes, T the set of terminal nodes, and P the set of transformation rules, also called grammar productions.

We follow the approach of Guennoun [Guennoun et al., 2004] that combines the structure of a Double PushOut (DPO) production structure and negative application conditions (NAC). An instance belonging to the graph grammar is a graph containing only terminal nodes obtained by applying a sequence of productions in P starting from axiom AX. P specifies the set of grammar productions that are of the form (L, K, R, NAC) where L, R, K and NAC are subgraphs. Such productions are considered applicable to a given graph G if a graph homomorphism from L to G exists without NAC. If a production is applicable, its application leads to the removal of the subgraph occurrence $(L \setminus K)$, the insertion of an isomorphous copy of the subgraph $(R \setminus K)$.

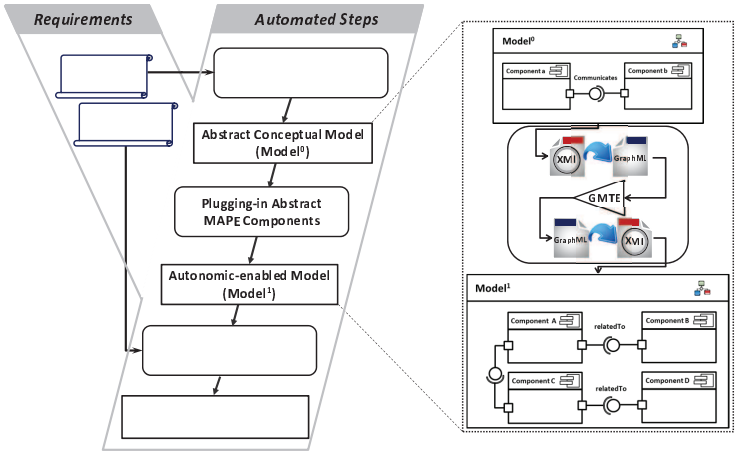

Figure 2: The transformation process inside a step

For each step, the initial architectural model is mapped automatically to XMI format (which is a widely used standard for exchanging information through XML) then to GraphML [Brandes et al., 2001] format (which is an XML format for graphs). The GMTE takes this GraphML file as an input. Then, it executes rules and transforms it into a refined architectural model. We note that the generated architectural model is correct by construction. The generated architectural model is a GraphML file and automatically mapped to XMI. Then, it can be represented as a UML diagram. Figure 2 illustrates an instance of a step.

3.2 Description of the Transformation Steps

3.2.1 Step 0: Initial Abstract Architectural Model (AAM)

In this step, the architect expresses, at a high level description, the functional requirements using an abstract architectural model that we denote $Model^0$. This model is handmade. It provides a description of structural requirements such as communications channels between the networked services and their composing software components. It represents an instantiation of an architectural style for SOA-compliant systems. Architectural styles are used to describe software architectures grouped by common resource types, configuration patterns and constraints [Abowd et al., 1993]. The proposed style is based on two types of component namely "Source" and "Destination", respectively equivalent to "Requester" and "Provider" in SOA. This architectural model ($Model^0$) is described as a UML component diagram.

The autonomic components will be included in the next steps. Therefore, we can consider "Equivalent destinations" for the "Original destinations" in order to perform the dynamic reconfiguration according to the reconfiguration action when a degradation occurs to the "Original destination". Such actions may include the substitution, the load balancing, etc. The "Equivalent destinations" are components that offer the same business logic as the original components.

3.2.2 Step 1: Autonomic Architecture Component Model (AACM) Generation

The goal of this step is to generate an autonomic architecture description model that extends the initial architecture description by the appropriate components that manage the behaviour of the system. It plugs in an instance of the autonomic control loop components between each "Destination" and its "Sources" on the abstract architectural model.

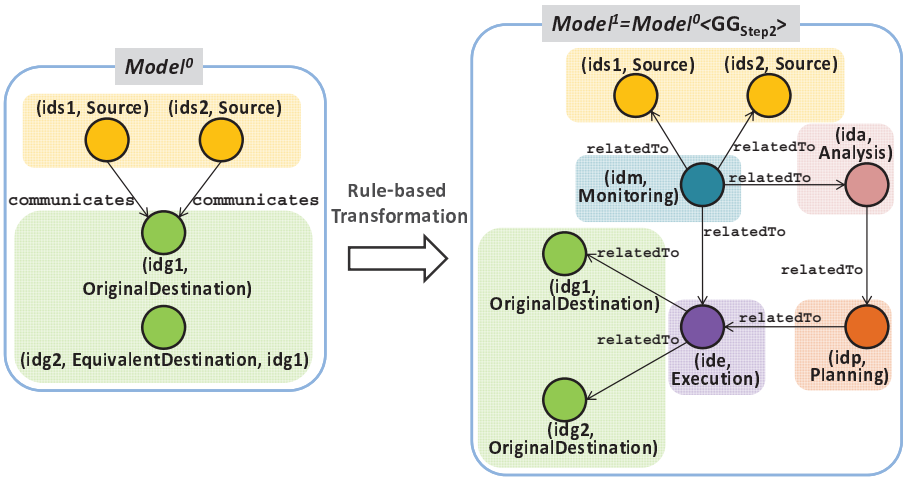

Figure 3: Application example of the GG_{Step1} graph grammar

We assume that the abstract architectural model $Model^0$ is composed of two "Sources", a "Original destination" and an "Equivalent destination". A graphical representation of the corresponding graph is shown in the left side of Figure 3. This architectural model is automatically mapped to XMI, then to GraphML. After that, we run the GMTE which takes this GraphML description as input. Our Engine applies the graph grammar, GG_{Step1}, described in Table 1 which

$GG_{Step1} = (AX, NT, T, P)$ with: $T = \{N_i(idComponent, typeComponent)\}$, $NT = \{(N_i, \text{“}EquivalentDestination\text{”}, idEquivalentComponent)\}$ and $P = \{p1, p2, p3\}$
$p1 =($ $L = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{”}OriginalDestination\text{”}),$ $N_1 \xrightarrow{\text{“}communicates\text{”}} N_2\};$ $K = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}OriginalDestination\text{”})\};$ $R = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}OriginalDestination\text{”}),$ $N_3(Id_3, \text{“}Monitoring\text{”}), N_4(Id_4, \text{“}Analysis\text{”}), N_5(Id_5, \text{“}Planning\text{”}),$ $N_6(Id_6, \text{“}Execution\text{”}), N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_1, N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_4,$ $N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_6, N_4 \xrightarrow{\text{“}relatedTo\text{”}} N_5, N_5 \xrightarrow{\text{“}relatedTo\text{”}} N_6,$ $N_6 \xrightarrow{\text{“}relatedTo\text{”}} N_2\};$ $NAC = \{N_6(Id_6, \text{“}Execution\text{”}), N_6 \xrightarrow{\text{“}relatedTo\text{”}} N_2\};)$
$p2 =($ $L = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}OriginalDestination\text{”}),$ $N_3(Id_3, \text{“}Monitoring\text{”}), N_4(Id_4, \text{“}Analysis\text{”}), N_1 \xrightarrow{\text{“}relatedTo\text{”}} N_2,$ $N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_4, N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_2\};$ $K = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}OriginalDestination\text{”}),$ $N_3(Id_3, \text{“}Monitoring\text{”}), N_4(Id_4, \text{“}Analysis\text{”}), N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_4,$ $N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_2\};$ $R = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}OriginalDestination\text{”}),$ $N_3(Id_3, \text{“}Monitoring\text{”}), N_4(Id_4, \text{“}Analysis\text{”}), N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_1,$ $N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_4, N_3 \xrightarrow{\text{“}relatedTo\text{”}} N_2\};)$
$p3 =($ $L = \{N_2(Id_2, \text{“}OriginalDestination\text{”}), N_6(Id_6, \text{“}Execution\text{”}),$ $N_7(Id_7, \text{“}EquivalentDestination\text{”}, Id_2), N_6 \xrightarrow{\text{“}relatedTo\text{”}} N_2\};$ $K = \{N_2(Id_2, \text{“}OriginalDestination\text{”}), N_6(Id_6, \text{“}Execution\text{”}),$ $N_6 \xrightarrow{\text{“}relatedTo\text{”}} N_2\};$ $R = \{N_2(Id_2, \text{“}OriginalDestination\text{”}), N_6(Id_6, \text{“}Execution\text{”}),$ $N_7(Id_7, \text{“}OriginalDestination\text{”}), N_6 \xrightarrow{\text{“}relatedTo\text{”}} N_2,$ $N_6 \xrightarrow{\text{“}relatedTo\text{”}} N_7\};)$

Table 1: Grammar productions for the generation of the AACM

incorporates effectively the autonomic control loop components. A graphical representation of the output graph is shown in the right side of Figure 3.

We use the following notations in the sequel: graph nodes are represented by $N_i(att_1, \cdots att_n)$ where "i" allows a node to be identified in the graph and where $att_1, \cdots att_n$ are attributes of the node. Attributes represent the uplet: ($idComponent$, $typeComponent$) if the node is terminal and ($idComponent$, $typeComponent$, $idEquivalentComponent$) if the node is non-terminal. The $typeComponent$ attribute defines the type of the component. It is defined as an enumeration type. This type is defined by the values of the set S_1 with S_1=$\{Source$, $OriginalDestination$, $EquivalentDestination$, $Monitoring$, $Analysis$, $Planning$, $Execution\}$. The attribute $IdEquivalentComponent$ is used when the type of the component ($typeComponent$) value is $EquivalentDestination$.

In Table 1, the production $p1$ substitutes the communication link between "Source" (N_3) and "OriginalDestination" (N_4) by an autonomic control loop -Monitoring, Analysis, Planning, Execution- (N_3, N_4, N_5, N_6).

Applying the production $p2$ substitutes the link between "Source" (N_1) and "OriginalDestination" (N_2) by a link between "Monitoring" (N_3) and "Source" (N_1). Connecting the monitoring node to the source enables the capture of necessary measurements that will be used by the analysis node.

Applying the production $p3$ transforms an "EquivalentDestination" on a "OriginalDestination" (N_7) and links the execution node N_6 ("Execution") to the new node. This production allows to the execution node the enforcement of reconfiguration plans that adapt the behavior of the application.

The application of GG_{Step1} is performed as follow: we apply the production $p1$ as long as possible. When it is not applicable, we apply the $p2$ as long as possible. When it is not applicable, we apply $p3$.

The autonomic architectural model resulting from this transformation is given as a GraphML description. It corresponds to the graph of $Model^1$ in the right side of Figure 3. This graph includes new components namely: Monitoring, Analysis, Planning, and Execution.

In $Model^0$, the nodes are linked through the label `communicates` that describes their interactions. In $Model^1$, all nodes are connected to each other via a new label namely `relatedTo`. A is `relatedTo` B means that A and B can exchange data flows or invocations.

Despite the transformation made on the abstract conceptual architectural model ($Model^0$), the autonomic architectural model still keeps the abstract view via the `relatedTo` label and the added autonomic abstract components. In the next subsection, we detail these abstract components regarding the non-functional requirements.

3.2.3 Step 2: Autonomic Architecture Deployment Model (AADM) Generation

In this step, we refine the content of autonomic control loop components regarding the non-functional requirements. We use the GMTE and graph grammars in order to refine the current architectural model. The input of this step is $Model^1$ that corresponds to the output of *step 1*. After transformation, we get the graph $Model^2$ shown in Figure 4.

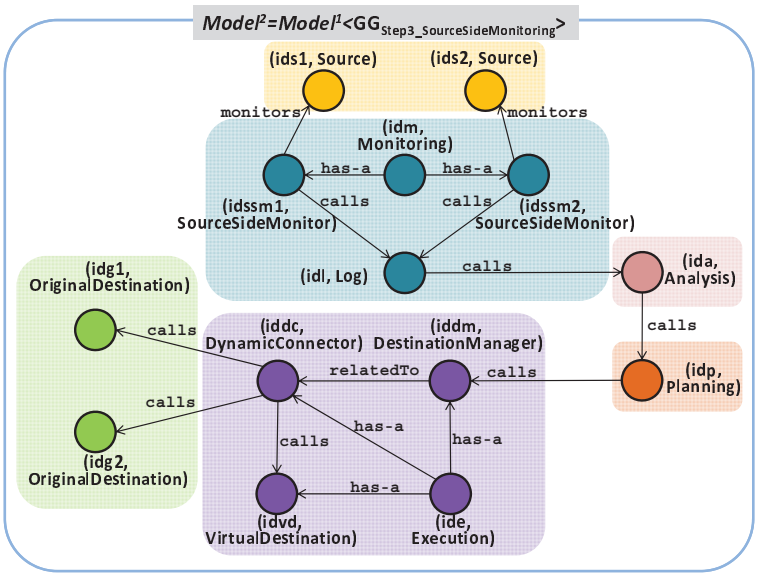

Figure 4: Application example of the $GG_{Step2\_SourceSideMonitoring}$ graph grammar

A first set of refinement graph grammar productions ($p1$ and $p2$ of $GG_{Step2\_SourceSideMonitoring}$) allows adding new nodes composing the Execution node. In our reconfigurable architectural model, the Execution node contains three children: "DestinationManager", "DynamicConnector" and "VirtualDestination". The "DestinationManager" enforces the dynamic reconfiguration plans. The "DynamicConnector" is the main node in the Execution. It binds the source requests, initially sent to the "VirtualDestination", to the "OriginalDestination". This is done according to the plan performed by the "DestinationManager" node. The "VirtualDestination" offers the same interfaces as the "OriginalDestination" with an empty body.

Therefore, a new label is added namely `has-a` and the `relatedTo` is refined to `calls`. A `has-a` B, means that the node A is a composite and B is a child of A. While, A `calls` B means that the node A invokes the node B.

The production $p3$ of $GG_{Step2\_SourceSideMonitoring}$ details the Monitoring node. The `relatedTo` links are refined to `monitors` links. A `monitors`

$GG_{Step2\_SourceSideMonitoring} = (AX, NT, T, P)$ with:
$T = \{N_i(idComponent, typeComponent)\}$,
$NT = \{\}$ and $P = \{p1, p2, p3\}$

$p1 =($
$L = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}Monitoring\text{”}), N_8(Id_8, \text{“}Analysis\text{”}),$
$N_9(Id_9, \text{“}Planning\text{”}), N_{10}(Id_{10}, \text{“}Execution\text{”}),$
$N_6(Id_6, \text{“}OriginalDestination\text{”}),$
$N_2 \xrightarrow{\text{“}relatedTo\text{”}} N_1, N_2 \xrightarrow{\text{“}relatedTo\text{”}} N_8,$
$N_9 \xrightarrow{\text{“}relatedTo\text{”}} N_{10}, N_{10} \xrightarrow{\text{“}relatedTo\text{”}} N_6,$
$N_2 \xrightarrow{\text{“}relatedTo\text{”}} N_{10}, N_8 \xrightarrow{\text{“}relatedTo\text{”}} N_9\};$
$K = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}Monitoring\text{”}), N_8(Id_8, \text{“}Analysis\text{”}),$
$N_9(Id_9, \text{“}Planning\text{”}), N_{10}(Id_{10}, \text{“}Execution\text{”}),$
$N_6(Id_6, \text{“}OriginalDestination\text{”})\};$
$R = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}Monitoring\text{”}), N_8(Id_8, \text{“}Analysis\text{”}),$
$N_9(Id_9, \text{“}Planning\text{”}), N_{10}(Id_{10}, \text{“}Execution\text{”}),$
$N_6(Id_6, \text{“}OriginalDestination\text{”}),$
$N_4(Id_4, \text{“}DynamicConnector\text{”}), N_7(Id_7, \text{“}VirtualDestination\text{”}),$
$N_3(Id_3, \text{“}Log\text{”}), N_{11}(Id_{11}, \text{“}DestinationManager\text{”}),$
$N_2 \xrightarrow{\text{“}monitors\text{”}} N_1, N_2 \xrightarrow{\text{“}calls\text{”}} N_3,$
$N_{10} \xrightarrow{\text{“}has-a\text{”}} N_4, N_{10} \xrightarrow{\text{“}has-a\text{”}} N_7, N_{10} \xrightarrow{\text{“}has-a\text{”}} N_{11},$
$N_3 \xrightarrow{\text{“}calls\text{”}} N_8, N_8 \xrightarrow{\text{“}calls\text{”}} N_9, N_9 \xrightarrow{\text{“}calls\text{”}} N_{11},$
$N_{11} \xrightarrow{\text{“}relatedTo\text{”}} N_4, N_4 \xrightarrow{\text{“}calls\text{”}} N_6, N_4 \xrightarrow{\text{“}calls\text{”}} N_7\};)$

$p2 =($
$L = \{N_4(Id_4, \text{“}DynamicConnector\text{”}), N_{10}(Id_{10}, \text{“}Execution\text{”}),$
$N_6(Id_6, \text{“}OriginalDestination\text{”}),$
$N_{10} \xrightarrow{\text{“}has-a\text{”}} N_4, N_{10} \xrightarrow{\text{“}has-a\text{”}} N_6\};$
$K = \{N_4(Id_4, \text{“}DynamicConnector\text{”}), N_{10}(Id_{10}, \text{“}Execution\text{”}),$
$N_6(Id_6, \text{“}OriginalDestination\text{”}),$
$N_{10} \xrightarrow{\text{“}has-a\text{”}} N_4\};$
$R = \{N_4(Id_4, \text{“}DynamicConnector\text{”}), N_{10}(Id_{10}, \text{“}Execution\text{”}),$
$N_6(Id_6, \text{“}OriginalDestination\text{”}),$
$N_{10} \xrightarrow{\text{“}has-a\text{”}} N_4, N_4 \xrightarrow{\text{“}calls\text{”}} N_6\};)$

$p3 =($
$L = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}Monitoring\text{”}), N_3(Id_3, \text{“}Log\text{”}),$
$N_2 \xrightarrow{\text{“}calls\text{”}} N_3,$
$N_2 \xrightarrow{\text{“}monitors\text{”}} N_1\};$
$K = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}Monitoring\text{”}), N_3(Id_3, \text{“}Log\text{”}),$
$N_2 \xrightarrow{\text{“}calls\text{”}} N_3\};$
$R = \{N_1(Id_1, \text{“}Source\text{”}), N_2(Id_2, \text{“}Monitoring\text{”}), N_3(Id_3, \text{“}Log\text{”}),$
$N_4(Id_4, \text{“}SourceSideMonitor\text{”})$
$N_2 \xrightarrow{\text{“}has-a\text{”}} N_4, N_2 \xrightarrow{\text{“}calls\text{”}} N_3,$
$N_2 \xrightarrow{\text{“}monitors\text{”}} N_1\};)$

Table 2: Grammar productions for the generation of the AADM

B means that the node A observes either the flows or the state of the node B. The graph grammars corresponding to this refinement depend on the non-functional requirements that define the QoS to monitor. The abstract monitoring component can be refined to three different models corresponding to three different graph grammars ($GG_{Step2\_SourceSideMonitoring}$, $GG_{Step2\_DestinationSideMonitoring}$, $GG_{Step2\_BothSidesMonitoring}$). Two types of nodes can be added,“SourceSideMonitor” and “DestinationSideMonitor”, depending on the needs of the architect.

Enabled Source Side Monitoring: This architectural model corresponds to monitoring only the source side. The monitor belongs only to the requester. For instance, it enables monitoring the response time. The associated graph grammar ($GG_{Step2\_SourceSideMonitoring}$) is described in Table 2.

Enabled Destination Side Monitoring: This architectural model (obtained using $GG_{Step2\_DestinationSideMonitoring}$) corresponds to monitoring only the destination side. We associate for all equivalent providers a single monitor. For instance, it enables monitoring the execution time. For sake of shortness, we do not give this graph grammar.

Enabled both Source and Destination Sides Monitoring: This architectural model (obtained using $GG_{Step2\_BothSidesMonitoring}$) corresponds to monitoring both source and destination sides. It deploys monitors in both sides. In addition to the response time and the execution time, it enables monitoring the communication time. For sake of shortness, we do not give this graph grammar.

In Table 2, a terminal node is represented by an uplet: $(idComponent, typeComponent)$. The $typeComponent$ attribute defines the component type. It is defined as an enumeration. This type is defined by the values of the set S_2 with $S_2 = \{Source, Execution, Monitoring, Log, Analysis,$

$Planning$, $VirtualDestination$, $OriginalDestination$, $SourceSideMonitor$, $DynamicConnector$, $DestinationManager$}.

The application of $GG_{Step2\_SourceSideMonitoring}$ is performed as follow: we apply the production $p1$ as long as possible. When it is not applicable, we apply the $p2$ as long as possible. When it is not applicable, we apply $p3$.

In the output architectural model of this step ($Model^2$), the abstract `relatedTo` labels is refined to `has-a`, `monitors` and `calls` that describe the interaction between nodes, and new nodes have been added that give a detailed view of the architectural model regardless of the platform.

Figure 4 shows the content the autonomic control loop components. In this case, only a source side monitor is deployed. Requests are sent to the "Virtual Destination", and then intercepted by the "Dynamic Connector" which reroutes them to the suitable "Original Destination". More details are available in [Emna Mezghani, 2013].

The resulting architectural model, which is autonomic enabled, can perform a dynamic reconfiguration when a QoS degradation occurs to recover the running application. Then, it is easily mapped to SOA: each component corresponds to a service. The implementation of this architectural model allows to get an autonomic Web Service based-application that is continuously adapted.

4 The Campus-Wide Smart Metering: an Application Use Case

In this section, we present our motivating example namely *Campus-Wide Smart Metering* (CWSM). Then, from this example, we present some transformations and refinements rules applied on this example in order to obtain a dynamic reconfigurable application.

4.1 Motivating Example

A considerable attention has been paid in recent years to the construction of smart buildings regarding their potential for reducing the power consumption. Whether newly designed, these buildings are equipped with heterogeneous devices (sensors, actuators, etc.), which are able to supervise and react to the context changes. Figure 5 depicts the CWSM which represents an example of M2M[2] applications. Indeed, the M2M application is one of the prominent SOA applications that are built on top of a network of services. Its architecture evolves three entities (Device, Gateway and Server). Each entity can provide or consume a service. The CWSM consists of a Remote Control Unit (RCU) and a number of smart buildings composed of conference rooms, labs, smart houses, etc.

[2] Machine to Machine, http://www.etsi.org/technologies-clusters/technologies/m2m

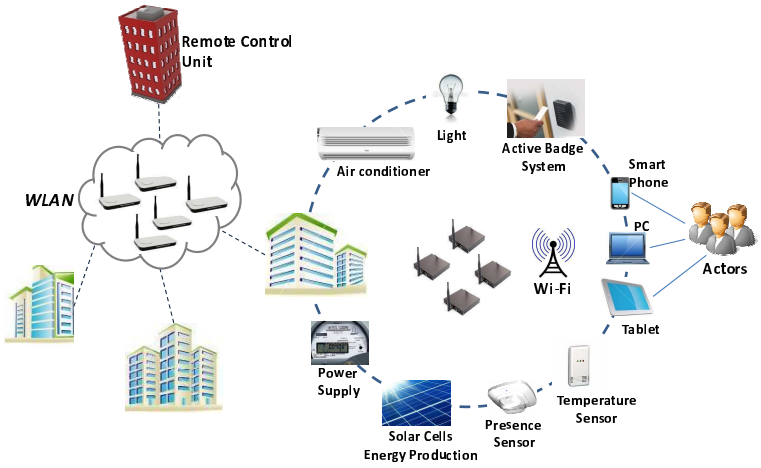

Figure 5: Campus-Wide Smart Metering use case

Each smart building includes heterogeneous devices equipped by sensors which supervise context parameters, and/or actuators that adjust them. Such devices can be air-conditioners, lamps, smart meters, access control badge systems, power-supply, and solar cells energy production. These cells track the solar radiations in order to convert sun-light into electricity. Sensors record information related to the environment such as rooms luminosity, human presence, temperature, etc. According to this information, actuators configure devices in order to promote and rationalize the efficient use of energy– for example, by turning off lights and air conditioning in unoccupied areas. This information is transmitted to the RCU via gateways. Regarding the pervasive nature of this environment, a number of challenges should be taken into account when designing such buildings. Various issues such as the device mobility and the bandwidth consumption should be considered. Indeed, devices may dynamically join and leave the environment in which they are located. So, gateways may reach an overload state that may lead to a system degradation expressed in terms of increasing the response time and in worst case, the request will be not delivered because of the *Time out connection*. For instance, if the requests sent from the *Presence Sensors* to the RCU are not well delivered on time the delivered QoS of our system will not be ensured such opening the light or turning on the air-condition of the room. Therefore, it is crucial to conceive an autonomic system at design time that dynamically reconfigures applications at runtime. In this context, our approach is located since it incorporates the dynamic reconfiguration to SOA-compliant applications. We illustrate the efficiency of our approach by considering the sensors as sources that generated data and the gateways as destinations that offer

the service "connect_to_RCU" in order to route these data to the RCU.

4.2 Implementation

In this section, we present the architectural models generated by our approach. The first architectural model ($Model^0$) is drawn by the architect in which he describes the business requirements. We used an enhanced version of the tool *G-Meidya* [Khlif et al., 2012] which is provided as an Eclipse plugin, used to portray software architectures. As described in Figure 6, we illustrate in this model the communication between the *Presence Sensor*, the *Light*, the *Gateway* already connected to them and an *Equivalent Gateway* that offers the same functionality.

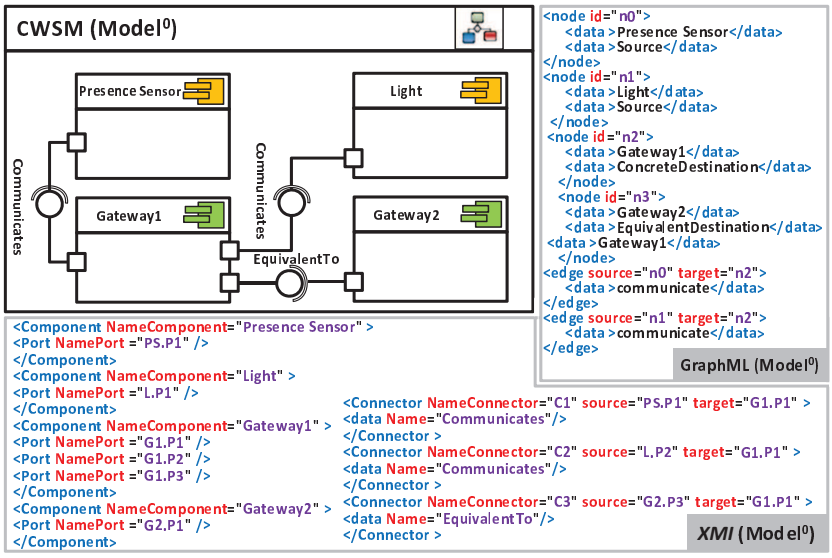

Figure 6: Model^0 of CWSM in three formats: UML, XMI and GraphML

The first transformation incorporates the autonomic control loop components. Therefore, the UML component diagram $Model^0$, which represents the entities system, will be generated in the XMI format thanks to *G-Meidya*. This file will be automatically transformed to the GraphML format in order to be the input to the GMTE as presented in Figure 6. The XMI file describes the business components and the communication connectors. While, each node in the GraphML file corresponds to a component and each edge is equivalent to a communication link.

As a first transformation, the graph grammar GG_{Step1} productions will be executed under GMTE. The final result of this graph grammar $Model^1$ is presented in Figure 7 in the UML, XMI and GraphML formats.

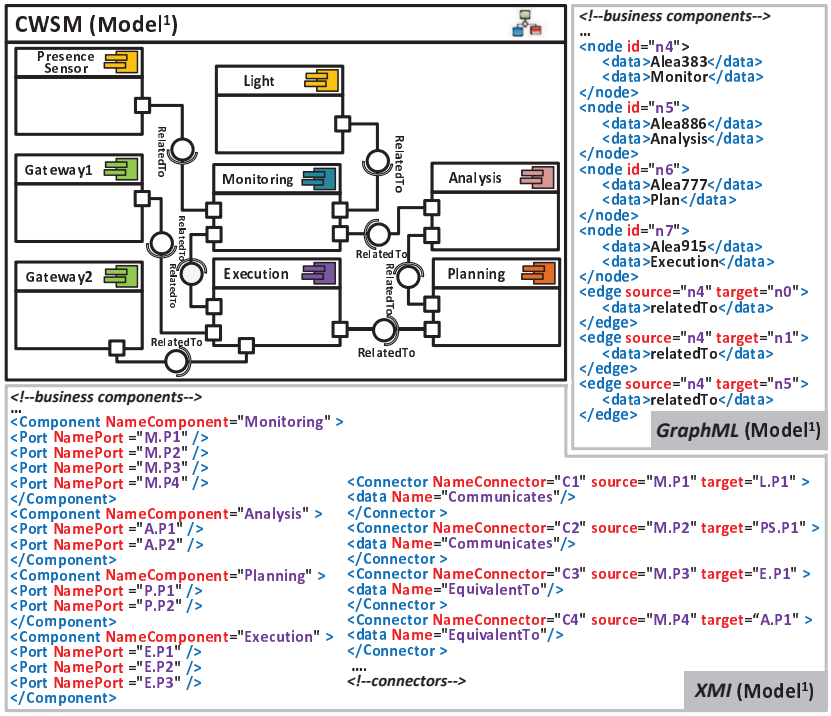

Figure 7: Model^1 of CWSM in three formats: UML, XMI and GraphML

Finally, after executing all the graph grammars corresponding to *step 1* and *step 2*, we generate the architectural model illustrated in Figure 8. It is an autonomic architecture able to dynamically reconfigure the application at runtime.

The *Source Side Monitors* observe QoS parameters. In this case, it corresponds to the response time between sensors and the gateway (*Gateway1*). Collected Values are saved in the Log. The *Analysis* component is notified about the availability of new data. Then, it detects possible QoS degradation. If detected (the *Gateway1* is overloaded), it asks the *Planning* component for a reconfiguration plan. The plan corresponds to a set of elementary reconfiguration actions. For instance, it may be the load balancing. Before performing the reconfiguration, all requests are sent to the *Virtual Destination* which reroutes them to the *Gateway1* then to RCU. When performing a dynamic reconfiguration, the *Destination Manager* provides and deploys a new *Dynamic Connector* which

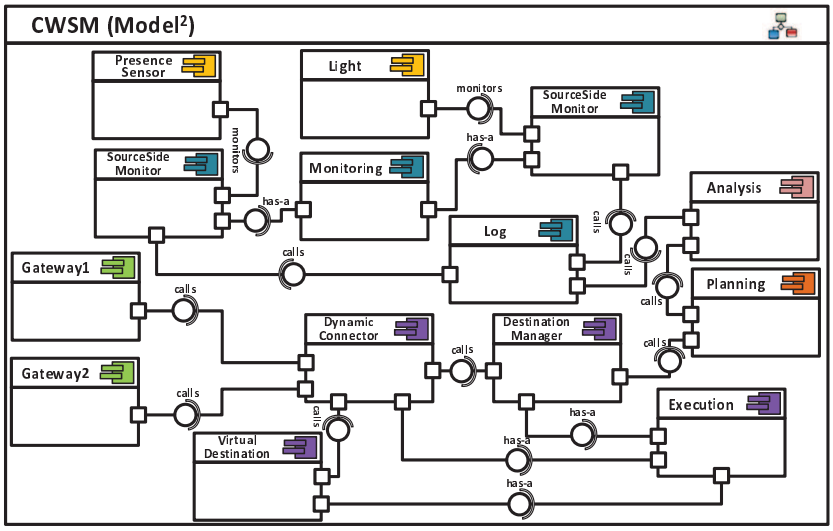

Figure 8: Model^2 of CWSM in UML

has the load balancing capability. The application will regain the correct state. After executing the reconfiguration, 50% of requests (from sensors) are sent to RCU through the *Gateway1* and 50% through the *Gateway2*. This will enhance the performance and avoid degradation.

An implementation of this architectural model with the Web Service technology has been already evaluated while using the "Load Balancing" as a reconfiguration action [Emna Mezghani, 2013].

5 Conclusion

Building dynamically reconfigurable architectures although important for a wide range of domains remains a task performed by hand. Autonomic architectures include the different components allowing a system to observe its state, to analyze it and to plan and execute reconfiguration actions. However, manually architecting autonomic systems is a hard task and may be error-prone. Therefore, it is important to provide an approach for automating their correct-by-design elaboration to satisfy requirements.

In this paper, we introduced a three-step approach that automates the transformation and the refinement of a given static architecture into a dynamically reconfigurable architecture. Without disturbing the functional requirement of the supported systems, the system architecture is enriched with new capabilities for managing the non-functional requirements at runtime. Endowing such capabilities is achieved using graph transformations and refinements rules.

Our approach hides application complexity and minimizes both design time and errors. The only effort considered from the architect side was to elaborate the initial abstract business model, and all transformations and refinements are automated. This approach is likely to be a helpful basis towards significant improvements on software scheduling and correctness, which may be especially useful for complex systems.

Our future work will mainly focus on representing or formalizing the non-functional requirements, currently described in XML format. Such formalization may include a semantic description using ontologies to reason and extend the power of the characterization of services.

Acknowledgments

This research is supported by the ITEA2's A2NETS (Autonomic Services in M2M Networks) project[3].

References

[Abowd et al., 1993] Abowd, G., Allen, R., and Garlan, D. (1993). Using style to understand descriptions of software architecture. *SIGSOFT Softw. Eng. Notes*, 18(5):9–20.

[Baresi et al., 2004] Baresi, L., Heckel, R., Thne, S., Varro, D., Varr, D., and Milano, P. D. (2004). Style-based refinement of dynamic software architectures. In *In Proc. 4 th Working IEEE/IFIP Conference on Software Architecture, WICSA4*, pages 155–164. IEEE.

[Brandes et al., 2001] Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., and Marshall, M. S. (2001). GraphML Progress Report. In *Graph Drawing*, pages 501–512.

[Cao et al., 2008] Cao, X.-X., Miao, H.-K., and Xu, Q.-G. (2008). Modeling and refining the service-oriented requirement. In *Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE '08, pages 159–165, Washington, DC, USA. IEEE Computer Society.

[Chomsky, 1956a] Chomsky, N. (1956a). Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124.

[Chomsky, 1956b] Chomsky, N. (1956b). Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124.

[Cristian Ruz and Sauvan, 2011] Cristian Ruz, F. B. and Sauvan, B. (2011). Flexible adaptation loop for component-based soa applications. In *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, pages 29–36.

[Emna Mezghani, 2013] Emna Mezghani, Riadh Ben Halima, K. D. (to appear in 2013). *DRAAS: Dynamically Reconfigurable Architecture for Autonomic Services*, chapter Web Services Foundations, page 24p. Springer.

[Ermel et al., 1999] Ermel, C., Rudolf, M., and Taentzer, G. (1999). Handbook of graph grammars and computing by graph transformation. chapter The AGG approach: language and environment, pages 551–603. World Scientific Publishing Co., Inc., River Edge, NJ, USA.

[3] A2Nets: Autonomic Services in M2M Networks, https://a2nets.erve.vtt.fi/, last visit in August 2012

[Gauvrit et al., 2010] Gauvrit, G., Daubert, E., and Andr, F. (2010). Safdis: A framework to bring self-adaptability to service-based distributed applications. In *SEAA'10: Proceedings of the 2010 36th EUROMICRO Conference on, Software Engineering and Advanced Applications*, pages 211 – 218. IEEE Computer Society.

[Gei et al., 2006] Gei, R., Batz, G., Grund, D., Hack, S., and Szalkowski, A. (2006). Grgen: A fast spo-based graph rewriting tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., and Rozenberg, G., editors, *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer Berlin Heidelberg.

[Guennoun et al., 2004] Guennoun, K., Drira, K., and Diaz, M. (2004). A proved component-oriented approach for managins dynamic software architectures. In *Proc. 7th iasted international conference on software engineering and application*, Marina Del Rrey, CA, USA.

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

[Khlif et al., 2012] Khlif, I., Kacem, M. H., and khalil Drira (2012). Une approche de description multi-chelles et multi points de vue pour les architectures logicielles dynamiques. In *La Confrence francophone sur les Architectures Logicielles (CAL)*.

[Maurel et al., 2010] Maurel, Y., Diaconescu, A., and Lalanda, P. (2010). Ceylon: A service-oriented framework for building autonomic managers. In *Engineering of Autonomic and Autonomous Systems (EASe), 2010 Seventh IEEE International Conference and Workshops on*, pages 3 –11.

[Miller and Mukerji, 2003] Miller, J. and Mukerji, J. (2003). Mda guide version 1.0.1. Technical report, Object Management Group (OMG).

[Pena et al., 2006a] Pena, J., Hinchey, M. G., and Sterritt, R. (2006a). Towards modeling, specifying and deploying policies in autonomous and autonomic systems using an aose methodology. In *Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems*, EASE '06, pages 37–46, Washington, DC, USA. IEEE Computer Society.

[Pena et al., 2006b] Pena, J., Hinchey, M. G., Sterritt, R., Ruiz-Cortes, A., and Resinas, M. (2006b). A model-driven architecture approach for modeling, specifying and deploying policies in autonomous and autonomic systems. In *Proceedings of the 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, DASC '06, pages 19–30, Washington, DC, USA. IEEE Computer Society.

[Rafe et al., 2009] Rafe, V., Rafeh, R., Fakhri, P., and Zangaraki, S. (2009). Using mda for developing soa-based applications. In *Proceedings of the 2009 International Conference on Computer Technology and Development - Volume 01*, ICCTD '09, pages 196–200, Washington, DC, USA. IEEE Computer Society.

[Riadh Ben-Halima, 2008] Riadh Ben-Halima, Khalil Drira, M. J. (2008). Survey a qos-oriented reconfigurable middleware for self-healing web services. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, volume 1, pages 104 – 111. IEEE Computer Society.

[Rodriguez et al., 2010] Rodriguez, I. B., Drira, K., Chassot, C., Guennoun, K., and Jmaiel, M. (2010). A rule-driven approach for architectural self adaptation in collaborative activities using graph grammars. *Int. J. Autonomic Comput.*, 1(3):226–245.

[Sancho, 2009] Sancho, G. (2009). Modlisation multi-niveau pour des systmes ubiquitaires collaboratifs. Congres de doctorants.

[Sheng et al., 2010] Sheng, Q. Z., Yu, J., and Dustdar, S. (2010). *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*. Chapman & Hall/CRC, 1st edition.

[Utomo, 2011] Utomo, W. H. (2011). Implementation of mda method into soa environment for enterprise integration. *IJCSI International Journal of Computer Science Issues*, 8(3):1694–0814.