Autonomic Management Policy Specification: from UML to DSML

Benoît Combemale, Laurent Broto, Xavier Crégut, Michel Dayde, Daniel Hagimont

Autonomic Management Policy Specification: from UML to DSML

Benoît Combemale, Laurent Broto, Xavier Crégut,
Michel Daydé, and Daniel Hagimont

Institut de Recherche en Informatique de Toulouse (UMR CNRS 5505)
Université de Toulouse. France.
first_name.last_name@irit.fr

Abstract. Autonomic computing is recognized as one of the most promizing solutions to address the increasingly complex task of distributed environments' administration. In this context, many projects relied on software components and architectures to provide autonomic management frameworks. We designed such a component-based autonomic management framework, but observed that the interfaces of a component model are too low-level and difficult to use. Therefore, we introduced UML diagrams for the modeling of deployment and management policies. However, we had to adapt/twist the UML semantics in order to meet our requirements, which led us to define DSMLs. In this paper, we present our experience in designing the Tune system and its support for management policy specification, relying on UML diagrams and on DSMLs. We analyse these two approaches, pinpointing the benefits of DSMLs over UML.

1 Introduction

Today's computing environments are becoming increasingly sophisticated. They involve numerous complex software that cooperate in potentially large scale distributed environments. These software are developed with very heterogeneous programming models which rely on their own specific configuration facilities. Moreover, software environments integrate components from different providers with proprietary management interfaces. Therefore, the management[1] of these software (installation, configuration, repair, etc.) is a much complex task which consumes a lot of human ressources.

A very promising approach consists in implementing administration features as an autonomic software. Such a software can be used to deploy and configure applications in a distributed environment. It can also monitor the environment and react to events such as failures or overloads and reconfigure applications accordingly and autonomously. Many works in this area have relied on a component model to provide such an autonomic system support [1,2,3]. The basic idea is to encapsulate the managed elements (legacy software) in software components (called wrappers) and to administrate the environment as a component architecture. We designed and implemented such

[0] This work is supported by the SCORWARE RNTL project (contract ANR-06-TLOG-017).

[1] we also use the term administration to refer to management operations.

a component-based autonomic management system (called Tune) and used it for the management of complex legacy software infrastructures.

However, we rapidly observed that the interfaces of a component model are too low-level and difficult to use. This led us to explore the introduction of higher level formalisms for all the administration tasks (wrapping, configuration, deployment, reconfiguration). Our main motivation was to hide the details of the component model we rely on and to provide a more abstract and intuitive specification interface (such specifications are called *management policies*). We mainly relied on UML diagrams for the modeling of management policies, but as we were not experts in modeling languages and were more focussed on middleware issues, we made use of UML diagrams which are very pragmatic and close to our needs, and we had to adapt/twist the UML semantics in order to meet our requirements. This naturally led us to define Domain Specific Modeling Languages (DSMLs) and their associated metamodels.

In this paper, we report on this experience which consisted in three steps: (1) designing a component-based autonomic system, (2) introducing management policy specification formalisms based on UML diagram and (3) defining several DSMLs for policy specification. And we motivate the transition between each step.

The rest of the paper is structured as follows. Section 2 describes the design and implementation of Tune, which provides a framework for component-based autonomic administration. Section 3 presents our experience in providing UML diagram based support for management policy specification. The DSML support introduced for policy specification is presented in Section 4. After an overview of related works in Section 6, we overview the lessons learned from these experiments in Section 5 and we conclude in Section 7.

2 An Autonomic Management System

2.1 J2EE Use Case

The Java 2 Platform, Enterprise Edition (J2EE) defines a model for developing web applications [4] in a multi-tiered architecture. Such applications are typically composed of a web server (e.g. Apache), an application server (e.g. Tomcat) and a database server (e.g. MySQL). Upon an HTTP client request, either the request targets a static web document, in which case the web server directly returns that document to the client; or the request refers to a dynamically generated document, in which case the web server forwards that request to the application server. When the application server receives a request, it runs one or more software components (e.g. Servlets, EJBs) that query a database through a JDBC (*Java DataBase Connection*) driver. Finally, the resulting information is used to generate a web document that is returned to the web client.

In this context, the increasing number of Internet users has led to the need of highly scalable and highly available services. To face high loads and provide higher scalability of Internet services, a commonly used approach is the replication of servers in clusters. Such an approach (Figure 1, *legacy layer*) usually defines a particular software component in front of each set of replicated servers, which dynamically balances the load among the replicas. Here, different load balancing algorithms may be used, e.g. Random, Round-Robin, etc.

This example is characteristic of the management of a distributed software infrastructure where very heterogeneous servers are distributely deployed, configured and interconnected in order to provide a global service. The management of the whole infrastructure can be very complex and requires a lot of expertise. Many files have to be edited and configured consistently. Also, failures or load peaks (when the chosen degree of replication is too low) must be treated manually.

2.2 Component-Based Autonomic Computing

Component-based management aims at providing a uniform view of a software environment composed of different types of servers. Each managed server is encapsulated into a component and the software environment is abstracted as a component architecture. Therefore, deploying, configuring and reconfiguring the software environment is achieved by using the tools associated with the used component-based middleware.

The component model we used in Tune is the Fractal model [5]. A Fractal component is a runtime entity that is encapsulated and has one or several interfaces (access points to a component that supports a finite set of methods). The signatures of interface can be described by a standard Java interface declaration. Components can be assembled to form a component architecture by binding components interfaces (different types of bindings exists, including local bindings and distributed RMI-like bindings). An (XML based) Architecture Description Language (ADL) allows describing an architecture and an ADL launcher can be used to deploy such an architecture. Finally, Fractal provides a rich set of control interfaces for introspecting (observing) and reconfiguring a deployed architecture, i.e. controlling components' attributes and bindings.

Any software managed with Tune is wrapped into a Fractal component which interfaces its administration procedures. Therefore, the Fractal component model is used to implement a management layer (Figure 1) on top of the legacy layer (composed of the actual managed software). In the management layer, all components provide a management interface for the encapsulated software, and the corresponding implementation (the wrapper) is specific to each software (e.g. the Apache web server in the case of J2EE). Fractal's control interfaces allow managing the element's attributes and bindings with other elements, and the management interface of each component allows controlling its internal configuration state. Relying on this management layer, sophisticated administration programs can be implemented, without having to deal with complex, proprietary configuration interfaces (generally configuration files), which are hidden in the wrappers. Here, we distinguish two important roles: (1) the role of the management and control interfaces is to provide a means for configuring components and bindings between components. It includes methods for navigating in the component-based management layer or modifying it to implement reconfigurations. (2) the role of the wrappers is to reflect changes in the management layer onto the legacy layer. The implementation of a wrapper for a specific software may also have to navigate in the component management layer, to access key attributes of the components and generate legacy software configuration files[2].

[2] e.g. for configuring an Apache, we need to access attributes from both the Apache component and the Tomcat components it is bound with.

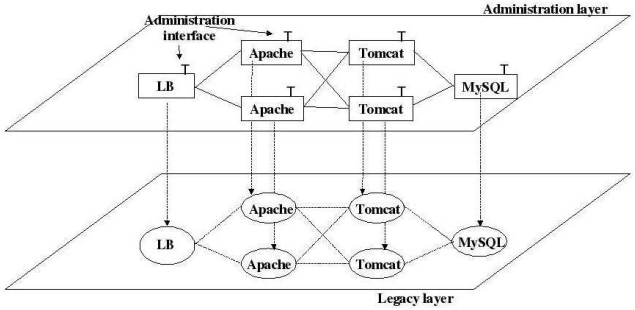

Fig. 1. Management layer for the J2EE application

2.3 Management Policy Specification

In a first prototype (Jade [2], a predecessor of Tune), the implementation of management policies was directly relying on the interfaces of the Fractal component model:

- A wrapper was implemented as a Fractal component, developed in Java, which main role is to reflect management/control operations onto the legacy software. For instance, if we consider the wrapper of the Apache software, the assignment of the port attribute of the wrapper is reflected in the *httpd.conf* file in which the port attribute is defined. Similarly, setting up a binding between an Apache wrapper and a Tomcat wrapper is reflected at the legacy layer in the *worker.properties* file.
- The description of a software architecture to be deployed was described in a Fractal ADL file. This ADL file describes in an XML syntax the set of components (wrappers) to instanciate (which will in turn deploy the associated legacy software components), their bindings and their configuration attributes.
- Reconfigurations were developed in Java, relying on Fractal APIs. These APIs allow invoking components' management interfaces or Fractal control interfaces for assigning components' attributes, adding/removing components and updating bindings between components.

Component-based autonomic computing has proved to be a very convenient approach. The experiments we conducted with this first prototype for managing J2EE infrastructures [2] (but also other distributed infrastructures such as Diet grid middleware [6]) validated this design choice. Figure 2 illustrates an experiment which consisted in automatically repairing (restarting) a failing Tomcat server (in the J2EE architecture of Figure 1). Initially, the load is balanced between the two replicas. When the failure occurs, all the load is addressed to the second replica. After repair, the load is again balanced between the two replicas.

3 UML-Based Autonomic Computing Policies Specification

As our system was used by external users (external to our group), we rapidly observed that the interfaces of a component model are too low-level and difficult to use. In order

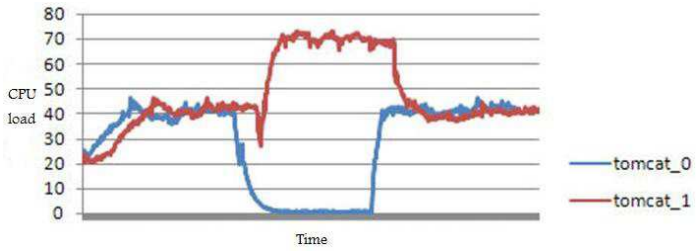

Fig. 2. Automatic restart of a Tomcat server on software fault

to implement wrappers (to encapsulate existing software), to describe deployed architectures and to implement reconfiguration programs, the administrator of the environment has to learn (yet) another framework, the Fractal component model in our case. More precisely, our previous experiments showed us that:

- wrapping components is difficult to implement. The developper needs to have a good understanding of the component model we use (Fractal). Regarding wrapping, our approach is to introduce a Wrapping Description Language which is used to specify the behavior of wrappers. A WDL specification is interpreted by a generic wrapper Fractal component, the specification and the interpreter implementing an equivalent wrapper. Therefore, an administrator doesn't have to program any implementation of Fractal component.
- architectures are not very easy to describe. ADLs are generally very verbose and still require a good understanding of the underlying component model. Moreover, if we consider large scale software infrastructure such as those deployed over a grid, describing an architecture composed of a thousand of servers requires an ADL description file of several thousands of lines. Our approach is to reuse UML formalisms for graphically describing architecture schemas. First, a UML based graphical description of such an architecture is much more intuitive than an ADL specification, as it doesn't require expertise of the underlying component model. Second, the introduced architecture schema is more abstract than the previous ADL specification, as it describes the general organisation of the application to deploy (types of software, interconnection pattern) in intension, instead of describing in extension all the software instances that may compose the architecture. This is particularly interesting for grid applications where thousands of servers have to be deployed.
- autonomic managers (reconfiguration policies) are difficult to implement as they have to be programmed using the management and control interfaces of the management layer. This also requires a strong expertise regarding the used component model. Our approach is to reuse UML State Diagrams to define workflows of operations that have to be performed for reconfiguring the managed environment. One of the main advantage of this approach, besides simplicity, is that state diagrams manipulate the entities described in the deployment schema and reconfigurations can only produce a concrete architecture which comforms to the abstract schema, thus enforcing reconfiguration correctness.

We detail these three aspects in the next sub-sections.

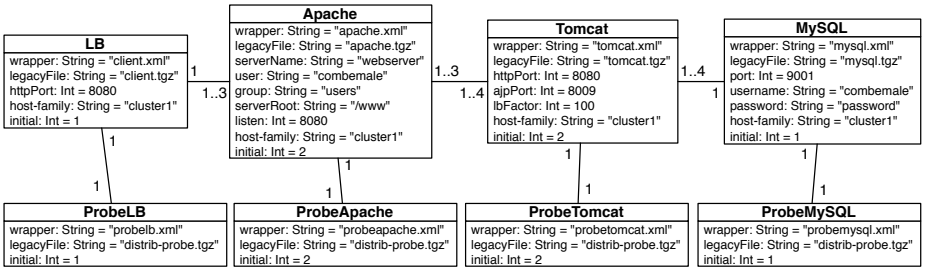

Fig. 3. Architecture schema for J2EE

3.1 UML-based formalism for architecture schemas

We adapted the UML class diagram formalism in order to allow specification of architecture schemas, as illustrated in Figure 3 where such a schema is defined for a J2EE organization. An architecture schema describes the overall organization of a software infrastructure to be deployed. At deployment time, the schema is interpreted to deploy a component architecture. Each element (the boxes) corresponds to a software which can be instantiated in several component replicas. A link between two elements generates bindings between the components instanciated from these elements. Each binding between two components is bi-directional (actually implemented by 2 bindings in opposite directions), which allows navigation in the component architecture in order to fetch any configuration attribute of the software infrastructure.

An element includes a set of configuration attributes for the software. Most of these attributes are specific to the software, but few attributes are predefined by Tune and used for deployment: *wrapper* gives the name of the WDL description of the wrapper, *legacyFile* gives the archive which contains the legacy software, *hostFamily* gives a hint regarding the dynamic allocation of the nodes where the software should be deployed, *initial* gives the number of instances which should be deployed.

The schema in Figure 3 describes a J2EE organization where one Load-Balancer, two Apaches, two Tomcats and one MySQL should be deployed. A probe is linked with each software, which monitors the liveness of the server in order to trigger a repair procedure. In this schema, a cardinality is associated with each link. It which constrains the interconnection of the deployed components. An intensional schema may be ambiguous, i.e. the actual deployed component architecture (bindings between components) will depend on the implemented deployment runtime. However, the user may describe a more extensional schema which will better fit his requirements.

The schema in Figure 3 deploys a component architecture as illustrated in Figure 1.

3.2 A Wrapping Description Language

Upon deployment, the above schema is parsed and for each element, a number of Fractal components are created. These components implement the wrappers for the deployed software, which provide control over the software. Each wrapper component is an instance of a generic wrapper which is actually an interpreter of a WDL specification.

A WDL description defines a set of methods that can be invoked to configure or reconfigure the wrapped software. The workflow of methods that have to be invoked in

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<wrapper name='apache'>
  <method name="start" class="wrapper.util.GenericStart" method="start_with_linux" >
    <param ... /> <param ... /> </method>

  <method name="configure" class="wrapper.util.ConfigurePlainText" method="configure">
    <param ... /> <param ... /> </method>

  <method name="addWorkers" class="wrapper.util.ConfigurePlainText" method="configure">
    <param name="config-file" value="conf/worker.properties" />
    <param name="worker.list" value="Tomcat.nodeName" /> </method>

  <method name="stop" class="appli.wrapper.util.GenericStop" method="stop_with_linux" >
    <param ... /> <param ... /> </method>
</wrapper>
```

Fig. 4. A WDL specification

order to configure and reconfigure the overall software environment is defined thanks to a formalism introduced in Section 3.3. Generally, a WDL specification (illustrated in Figure 7) provides *start* and *stop* operations for controlling the activity of the software, and a *configure* operation for reflecting the values of the attributes (defined in the UML architecture schema) in the configuration files of the software. Notice that the values of these attributes can be modified dynamically. Other operations can be defined according to the specific management requirements of the wrapped software, these methods being implemented in Java.

The main motivation for the introduction of WDL are (i) to hide the complexity of the underlying component model (Fractal) and (ii) that most of the needs should be met with a finite set of generic Java methods implementations (that can be therefore reused). We covered the needs of our usecases with very few methods, plaintext and XML file accessors and a shell command launcher. A method definition includes the description of the parameters that should be passed when the method is invoked. These parameters may be String constants, attribute values or combinaison of both (String expressions). All the attributes defined in the architecture schema can be used to pass the configured attributes as parameters of the method invocations.

It is sometimes necessary to navigate in the deployed component architecture in order to configure the software. For instance, in a J2EE architecture, an Apache may be bound to several Tomcats. At the legacy layer level, the *worker.properties* configuration file of Apache must include the list of the nodes where the Tomcats have been launched. Therefore in Figure 7, the *addWorkers* method in the Apache wrapper must receive this list of nodes in order to configure the *worker.properties* file. The syntax of the method parameters in the wrapper allows navigating in the management layer in order to access the component attributes (in this case the *nodeName* attribute of each Tomcat), following the bindings between the Apache component and the Tomcat components. *Tomcat.nodeName* returns the list of *nodeName* attributes of the Tomcat components which are bound with the current Apache component.

3.3 UML-based formalism for (re)configuration procedures

Reconfigurations are triggered by events. An event can be generated by a specific monitoring component (e.g. probes in the architecture schema) or by a wrapped legacy software which already includes its own monitoring functions.

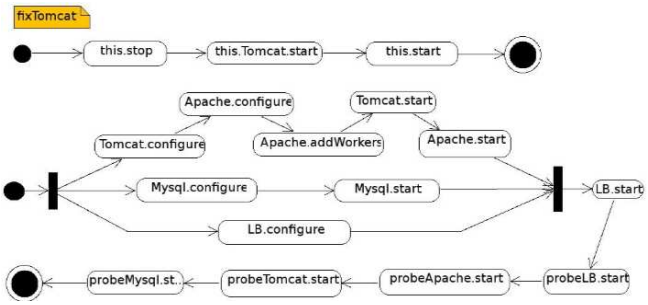

Fig. 5. State diagrams for repair and start

Whenever a wrapper component is instanciated, a communication pipe is created (typically a Unix pipe) that can be used by the wrapped legacy software to generate an event, following a specified syntax which allows for parameter passing. Notice that the use of pipes allows any software (implemented in any language environment such as Java or C++) to generate events. An event generated in the pipe associated with the wrapper is transmitted to the administration node where it can trigger the execution of reconfiguration programs (in our current prototype, the administration code, which initiates deployment and reconfiguration, is executed on one administration node, while the administrated software is managed on distributed hosts). An event is defined as an event type, the name of the component which generated the event and eventually an argument (all of type String).

For the definition of reactions to events, we reused the UML state diagrams formalism which allows specifying reconfiguration. Such a state diagram defines the workflow of operations that must be applied in reaction to an event. An operation in a state diagram can assign an attribute or a set of attributes of components, or invokes a method or a set of methods of components. To designate the components on which the operations should be performed, the syntax of the operations in the state diagrams allows navigation in the component architecture, similarly to the wrapping language.

For example, let's consider the diagram in Figure 5 (on the top) which is the reaction to a Tomcat (software) failure. The event (*fixTomcat*) is generated by a *probeTomcat* component instance, therefore the *this* variable references this *probeTomcat* component instance. Then:

- *this.stop* will invoke the *stop* method on the probing component (to prevent the generation of multiple events),
- *this.Tomcat.start* will invoke the *start* method on the Tomcat component instance linked with the probe. This is the actual repair of the faulting Tomcat server,
- *this.start* will restart the probe associated with the Tomcat.

Notice that state diagram's operations are expressed using the elements defined in the architecture schema, and are applied on the actually deployed component architecture. The current version of Tune also provides operations which re-deploy components (change location or add component instances) while enforcing the defined abstract architecture schema.

A particular diagram is used to start the deployed J2EE environment, as illustrated in Figure 5 (on the bottom). In this diagram, when an expression starts with the name of

an element in the architecture schema (Apache, Tomcat...), the semantics is to consider all the instances of the element, which may result in multiple method invocations. The starting diagram ensures that (1) configuration files must be generated, then (2) the servers must be started following the order Tomcat, Apache and LB (no constraint on MySQL). For each type of server, the server is started before its probe.

4 DSML-Based Autonomic Computing Policies Specification

Our first experiments with Tune focussed on the use of XML and UML to take advantage of well-known paradigms and of many existing open source tools. We used the UML2.0 graphical editors provided by the TOPCASED Eclipse-based toolkit [7] for the description of architectures and reconfiguration diagrams. However, the use of this unified language led us to specialize (pragmatically but sometimes awkwardly) its initial semantics in order to adapt it according to our needs. Because it is difficult to take into account this semantics specialization at the tools level, the user is let with all the freedom offered by UML.

For this reason, we are currently studying the possibility to define a dedicated metamodel for management policies definition. This allows us to define a constrained abstract syntax and a dedicated concrete syntax, relying on generic or generative tools, such as TCS [8] or Syntaks [9] for textual formalisms and TOPCASED [7] or GMF [10] for graphical formalisms. For each point of view that we have taken into account in Tune, we present the corresponding metamodel, offering a constrained, domain-specific and user-friendly languages.

4.1 The Configuration Description Language

The first language is the homogeneous definition of the application architecture. The UML-based formalism introduced in Section 3.1 was very close to the UML class diagram (rather than the UML component diagram, which can be quite confusing), reusing the concepts of classes, attributes and associations with multiplicities. However, the objective was mainly to reuse the expressiveness of the graphical notation, but with a domain specific semantic. The DSML we introduce here proposes a simple intentional architecture description language which allows to reify the heterogenous structural architecture of the legacy level. We call this language the *Configuration Description Language* (CDL). The main subset of the metamodel is depicted in Figure 6, also with an illustration with the model of the J2EE example of Section 3.

The main concept of this view is the *SoftwareElement* describing a particular type of software with its own configuration, management, and life cycle procedures. Each *SoftwareElement* is described by a set of properties (*ownedAttributes*), with an initial value (*defaultValue*), which are used by the administrator to reify the configurable attributes of the legacy software that the *SoftwareElement* represents. Note that a particular software can be reified by different *SoftwareElements* with different configuration properties.

The configuration language allows to describe an architecture in intension. This means here that each described *SoftwareElement* can be deployed into several instances. The architecture of the legacy level is intentionally reified through the definition of

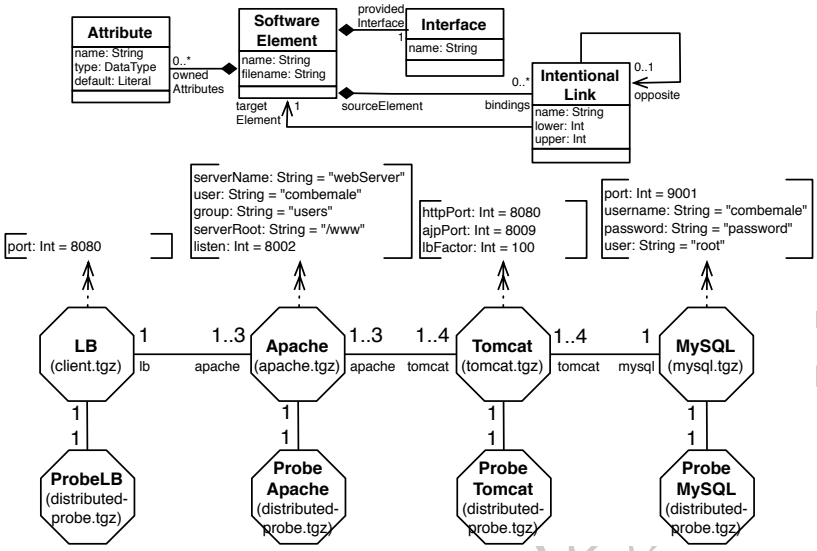

Fig. 6. The *Configuration Description Language*

bindings (*IntentionalLink*), allowing to connect a *SoftwareElement* to another, and expressing a multiplicity (*lower* & *upper*) and a role (*name*). The multiplicity expresses the range of instances of the target *SoftwareElement* for each one of the source *SoftwareElement*. The role allows navigation with a query language relying on OCL [11]. It is also possible to define bi-directional *bindings* by defining an *opposite bindings*.

4.2 The Wrapping Description Language

The second language allows the definition of a wrapper and its relation with *SoftwareElements*. In the Wrapping Description Language introduced in Section 3.2, a wrapper was described in an XML dialect, only enabling runtime checks by the Tune machinery. With this DSML, it becomes possible to introduce static consistency checks regarding the architecture schema it may reference, especially for the navigation clauses included in method parameters.

The corresponding metamodel is presented in Figure 7, also with an illustration with a model defined with a specialized textual editor. A *Wrapper* describes *methods* which define actions that can be applied on the encapsulated software component. A wrapper may be referenced by different *SoftwareElements* (with different properties). A *Method* can be parametrized (*ownedParameter*) with any property (of the *SoftwareElement*) of the configuration description in which the wrapper is used, the OCL-based navigation language allowing to fetch the effective parameter values. The method implementations (*imp*) are given in the form of a reference to a program (currently a string referring to a Java class).

Note that this view must be consistent with the architectural view described with the CDL. We have thus defined OCL constraints to verify that the wrapper associated with a software element defines at least the methods provided by the interface.

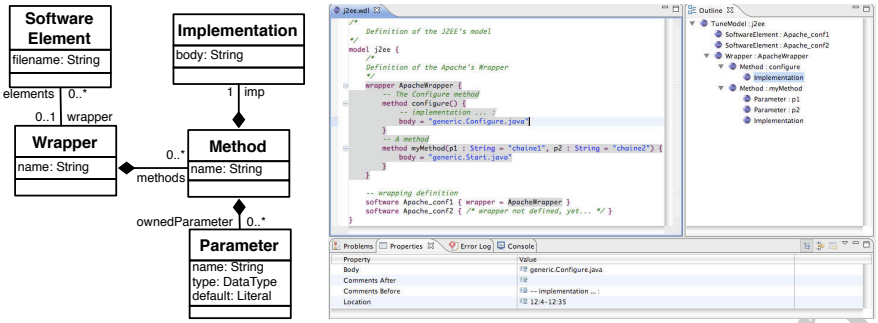

Fig. 7. The *Wrapping Description Language*

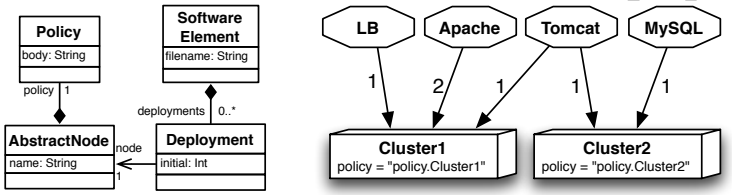

Fig. 8. The *Deployment Description Language*

4.3 The Deployment Description Language

The third language is used to define by intention or by extension, the real deployment of instances of each software component on system's nodes. In the UML-based formalisms described in Section 3.1, deployment policies were specified thanks to the *initial* and *host-family* attributes in the Architecture schema.

The introduced deployment DSML is described by the metamodel presented in Figure 8, also with the illustration with the J2EE example. For this, we define for each *SoftwareElement* a set of *Deployments*, describing a real number of instances (*initial*) to be deployed on a node (*AbtractNode*). Nodes are known as "abstract" because they define a deployment policy (*policy*). Abstract nodes include the deployment information required to implement a deployment strategy, e.g. the physical address of a (single) real node on which instances should be deployed, or a list of physical addresses and an allocation function (for a cluster).

Note that this view must be consistent with the view described with the CDL. For instance, the number of deployed instances must be compatible with the multiplicities described in the configuration.

The clear separation of the deployment and architecture concerns allows to define several deployment orders for the same *SoftwareElement* (i.e. on different nodes), and to define different deployment models for the same architecture model. Finally, we clearly identify the concept of node.

4.4 The Reconfiguration Description Language

The last DSML allows the definition of reconfiguration policies. In the UML-based formalism introduced in Section 3.3, reconfiguration actions were expressed in terms of

state diagrams, where we awkwardly inserted elementary actions as the state name. In accordance with the UML semantics, we decided to represent reconfiguration actions as activity diagrams in this DSML. Moreover, we introduced support in the DSML for the life cycle definition, allowing a clear description of links between events and reconfiguration actions, similarly to ECA (*Event/Condition/Action*) rules which typically used. This life cycle is expressed as a state diagram, where transitions are trigerred by events and execute reconfiguration actions defined as activity diagram.

We are inspired by the UML metamodel to express state diagrams (i.e. life cycles) and activity diagrams (i.e. reconfiguration policies). We don't reproduce this DSML metamodel since it is very similar to that of UML (with useless elements withdrawn).

5 Lessons Learned

5.1 Users'viewpoint

Dealing with autonomic management policies, we have experienced the definition of **higher level formalisms** than the one used by the underlying Fractal components. The main motivation is to provide a formalism easier to understand and facilitating the definition of the views describing an application. These views are parsed by the Tune runtime in order to configure a Fractal component architecture. The first main benefit of this approach is to ease the learning and the adoption of the tool, for new users but also for experienced users. As an example, one student with no preliminary knowledge has been able to deploy his first application within two hours after a one hour seminar whereas several days were previously required

In a first step, we have built our views by using UML diagrams, thus being close to a well-known and largely-adopted notation. Furthermore, it was possible to use the numerous available UML tools, providing the users with high level tools. Nevertheless, reusing the UML notation had one important drawback: we had to tailor the semantics of the diagrams we reused (the class and statemachine diagrams). Thus, we deviated from a standard UML model. For example, we used the name of a state to describe a method call rather that the UML behavior concept.

One solution, would have been to define a real UML profile rather than only reusing the graphical notation and tools but we would have also inherited the complexity of the UML metamodel. So, we investigated the use of DSMLs. It allows focussing on only the domain specific concepts and to define adapted views (as an example, the UML architecture schema has been split into two views, the configuration description language and the deployment description language). The semantics of the notation is well-founded because it is defined by a (MOF) metamodel completed by a set of OCL constraints to express constraints that are not captured by the metamodel itself.

Obviously, UML tools are not usable with our newly-defined DSMLs. So we had to develop new editors. The TOPCASED tool [7] has been used to generate the graphical editors. When sticking at the possibilities of this editor generator, some hours are sufficient to have a functional editor. If additional functionalities are required, the generated code has to be manually adapted. These adaptations may be time consuming but the base editor is really usable as is. We have also used TCS [8], a tool that permits

to define a concrete syntax and the associated Eclipse editor (including colour, folding, name completion, etc). So we have been able to provide in a couple of hours a textual syntax for WDL that is easier to use that the XML like syntax.

Finally, the graphical and textual generated editors provide better **user assistance** than the UML editors because they **enforce the construction of coherent models** conforming to the DSML definition, including OCL constraints. So, users can see their mistakes before the Tune runtime reports them.

5.2 Correctness

The adopted approach favors correctness of managed applications for 3 main reasons:

- the definition of the management layer. Thanks to the management layer (composed of wrappers), the administrator of a software infrastructure does not have to manipulate complex configuration files, as in a J2EE clustered architecture. All the configurable entities (attributes, bindings, etc.) are reified in the management layer and can be homogeneously manipulated.
- the definition of an application pattern and its enforcement. The deployed application architecture is generated from the definition of a pattern (the architecture schema). Any reconfiguration described with the Reconfiguration Description Language can only result in an architecture which complies with this pattern.
- the definition of a user-friendly notation that favors the understanding of the users on their models is another step in the right direction. Indeed, the users are enforced to build coherent models, especially in the DSML approach, or are informed of possible mistakes without having to wait until their models are parsed by the Tune runtime. The application is then automatically generated.

6 Related Works

Autonomic computing is an appealing approach that aims at simplifying the hard task of system management, thus building self-healing, self-tuning, and self-configuring systems [12]. Management solutions for legacy systems are usually proposed as ad-hoc solutions that are tied to particular legacy system implementations (e.g. [13] for self-tuning cluster environments). This unfortunately reduces reusability and requires autonomic management procedures to be reimplemented each time a legacy system is taken into account in a particular context. Moreover, the architecture of managed systems is often very complex (e.g. multi-tier architectures), which requires advanced support for its management.

Relying on a component model for managing legacy software infrastructure has been investigated by several projects [1,2,3] and has proved to be a very convenient approach, but in most cases, the autonomic policies have to be programmed using the programming interface of the underlying component model (a framework for implementing wrappers, configuration APIs or deployment ADLs) which is too low level and still error prone.

Therefore, many projects explored model-driven approaches for designing autonomic management policies. Some of them proposed frameworks for modeling autonomic systems, e.g. a self-healing [14], a self-protecting [15], or a resource management system [16], the management system implementation being generated from the described management model. The modeling of such a system can still be quite complex and the integration within a legacy software organisation tricky. Some other projects proposed frameworks and runtimes for modeling the managed system and maintaining consistency between the managed system and its model at runtime [17,18]. The main advantage is well defined representation of the managed system, on which management policies can be applied. The tune system falls into this category, even if our management layer relies on the Fractal component model. Tune relies on DSML for the specification of a software architecture, its deployment and reconfiguration. These languages ensure that only consistent system states can result from deployment and reconfiguration.

Finally, some projects considered interactions between policies, mainly in order to deal with conflicts [19,20]. We are currently working on a DSML which should allow such coordinating between reconfiguration policies.

7 Conclusion and Perspectives

We are investigating the design and implementation of an autonomic system called Tune. Tune relies on a component model in order to administrate a legacy software infrastructure as a component architecture. Tune provides support for encapsulating (wrapping) software, describing the software architecture to manage and its deployment in a physical environment, and describing the dynamic reconfiguration policies to be applied autonomously. Our experiments with Tune led us to the conclusion that higher-level support was required for assisting administrators in policy description tasks.

For this purpose, our first experiments focused on the use of UML-based formalisms. These experiments confirmed the interest of raising up the abstraction level but we had to specialize the UML semantics according to the requirements of the considered field. It was difficult to take into account this specialization in the tools we reused. In a second step, we worked on the definition of a dedicated metamodel. The expected benefits are two-fold: to provide a formal definition of Tune's languages, and to statically and dynamically validate the policies described by administrators with customized editors.

This work opens many perspectives on which we are currently working. Although we already prototyped few specialized editors, we plan to provide editing tools for all the administration points of view considered by Tune. Also, the Tune developpers are currently extending the reconfiguration capabilities of Tune and the metamodel and the associated editing tools will evolve accordingly. In the longer term, we plan to revisit the design of the Tune system, considering that the management layer (illustrated in Figure 1) should be managed as a model (instead of a component architecture at the middleware level). This means that models would not only be used to describe policies, but would further be used to maintain the internal state of the Tune system. We refer to this new MDE field as *Model-Driven System Administration*. We are convinced that it is now essential to increase the abstraction level of software management, not only during the design but also during their development and administration.

References

1. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self adaptation with reusable Infrastructure. In: IEEE Computer, 37(10). (2004)
2. Hagimont, D., Bouchenak, S., Palma, N.D., Taton, C.: Autonomic Management of Clustered Applications. In: IEEE International Conference on Cluster Computing. (2006)
3. Oriezy, P., Gorlick, M., Taylor, R., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., A.Wolf: An Architecture-Based Approach to Self-Adaptive Software. In: IEEE Intelligent Systems 14(3). (1999)
4. Microsystems, S.: Java 2 Platform Enterprise Edition (J2EE). `http://java.sun.com/j2ee/`
5. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal Component Model and its Support in Java. In: Software - Practice and Experience, special issue on *Experiences with Auto-adaptive and Reconfigurable Systems*. (September 2006)
6. Toure, M., Berhe, G., Stolf, P., Broto, L., Depalma, N., Hagimont, D.: Autonomic Management for Grid Applications. In: 16th Euromicro International Conference on Parallel, Distributed and network-based Processing. (February 2008)
7. The TOPCASED Project: `http://www.topcased.org/`
8. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, ACM (October 2006) 249–254
9. Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-driven analysis and synthesis of concrete syntax. In: Proceedings of the 9th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems. Volume 4199 of LNCS., Springer (October 2006) 98–110
10. GMF: Graphical Modeling Framework. `http://www.eclipse.org/gmf/`
11. Object Management Group: UML Object Constraint Language (OCL) 2.0. (June 2005)
12. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. In: IEEE Computer Magazine, 36(1). (2003)
13. Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P.: Dynamic Provisiong of Multi-Tier Internet Applications. In: 2nd International Conference on Autonomic Computing. (June 2005)
14. Jiang, M., Zhang, J., Raymer, D., Strassner, J.: A Modeling Framework for Self-Healing Software Systems. In: Workshop "Models@run.time" at the 10th International Conference on model Driven Engineering Languages and Systems. (2007)
15. Pena, J., Hinchey, M.G., Sterritt, R., Ruiz-Cortes, A., Resinas, M.: A model-driven architecture approach for modeling, specifying and deploying policies in autonomous and autonomic systems. In: 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing, IEEE Computer Society (2006) 19–30
16. Barrett, K., Davy, S., Strassner, J., Jennings, B., van der Meer, S., Donnelly, W.: A model based approach for policy tool generation and policy analysis. In: Proc. 1st IEEE Global Information Infrastructure Symposium, IEEE (2007) 99–105
17. Sriplakich, P., Waignier, G., Le Meur, A.F.: Enabling Dynamic Co-Evolution of Models and Runtime Applications. In: 1st IEEE International Workshop on Model-Driven Development of Autonomic Systems, IEEE Computer Society (July 2008)
18. Rohr, M., Boskovic, M., Giesecke, S., Hasselbring, W.: Model-driven Development of Self-managing Software Systems. In: Workshop "Models@run.time" at the 9th International Conference on model Driven Engineering Languages and Systems (MoDELS). (2006)
19. Agrawal, D., Lee, K.W., Lobo, J.: Policy-based management of networked computing systems. Communications Magazine, IEEE **43**(10) (October 2005) 69–75
20. Davy, S., Barrett, K., Serrano, M., Strassner, J., Jennings, B., van der Meer, S.: Policy Interactions and Management of Traffic Engineering Services Based on Ontologies. Network Operations and Management Symposium (September 2007) 95–105