Towards a Model Driven Autonomic Management System

Laurent Broto, Daniel Hagimont, Estella Annoni, Benoit Combemale and Jean-Paul Bahsoun
IRIT
Université Paul Sabatier, 118 Route de Narbonne
F-31062 TOULOUSE CEDEX 9
Email: first.last@irit.fr

***Abstract*— Distributed software environments are increasingly complex and difficult to manage, as they integrate various legacy software with specific management interfaces. Moreover, the fact that management tasks are performed by humans leads to many configuration errors and low reactivity. This is particularly true in medium or large-scale distributed infrastructures.**

To address this issue, we explore the design and implementation of an autonomic management system. The main principle is to wrap legacy software pieces in components in order to administrate a software infrastructure as a component architecture. However, we observed that the interfaces of a component model are too low-level and difficult to use. Consequently, we explore the use of a model driven approach where several UML profiles are used to specify the different facets of an autonomic management policy.

Keywords: autonomic computing, administration, component architectures, model-driven engineering.

I. 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 and their configuration facilities are generally proprietary. Therefore, the management[1] of these software (installation, configuration, tuning, repair ...) is a much complex task which consumes a lot of human ressources. A very promising approach to this issue is to implement administration 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 and to administrate the environment as a component architecture. Then, the administrators can benefit from the essential features of the component model, encapsulation, deployment facilities and reconfiguration interfaces, in order to implement their autonomic management processes. However, following this approach, we observed that the interfaces of a component model are too low-level and difficult to use. In order to implement wrappers (which interfaces with existing software), to describe deployed architectures and to implement reconfiguration programs, the administrator of the environment has to learn (yet) another component-based framework.

Therefore, we propose to rely on a model driven approach, and to hide the specificities and complexities of the underlying component model. In the system we developed called Tune, for each administration facet (deployment, configuration, reconfiguration), we introduce a UML-based [4], [5] Domain Specific Modeling Language (DSML) for the specification of the administration policies. We also introduce a simple description language for simplifying the encapsulation of software.

The rest of the paper is structured as follows. Section 2 presents the context of our work. Section 3 describes our contributions. After an overview of related works in Section 4, we conclude in Section 5.

II. Context

In this section, we first present the application context that we consider in order to illustrate our contributions. We then present in more details what we mean by component-based autonomic management.

A. Applications

Our main application target is the administration of servers distributed over a cluster of machines or a grid infrastructure. We conducted many experiments with clustered J2EE applications, which implement dynamic web servers executing over a cluster of machines. In this paper, we will rely on another example to illustrate our work, a load balancer distributed over a grid.

Grid computing aims at enabling the sharing, selection, and aggregation of geographically distributed resources, dynamically at runtime depending on their availability, capability, cost and user's quality of service requirements. Diet [6] is an example of middleware environment which aims at balancing computation load over a grid. It is built on top of different tools which are able to locate an appropriate server depending on the client requested function, the data location (which can be anywhere on the system, because of previous computations) and the dynamic performance characteristics of the system. The aim of Diet is to provide transparent access to a pool of computational servers at a very large scale. As illustrated

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

in Figure 1, Diet mainly has the following components. A client is an application which uses DIET to solve problems. A Master Agents (MA) receive computation requests from clients. Then a MA chooses the best server and returns its reference to the client. The client then sends the computation request to that server. Local Agents (LA) aim at transmitting monitoring information between servers and MAs. LAs don't take scheduling decision, but allow preventing MAs overloads when dealing with large-scale infrastructures. Server Daemons (SeD) encapsulate computational servers (processors or clusters). A SeD declares the problems it can solve to its parent LA and provides an interface to clients for submitting their requests.

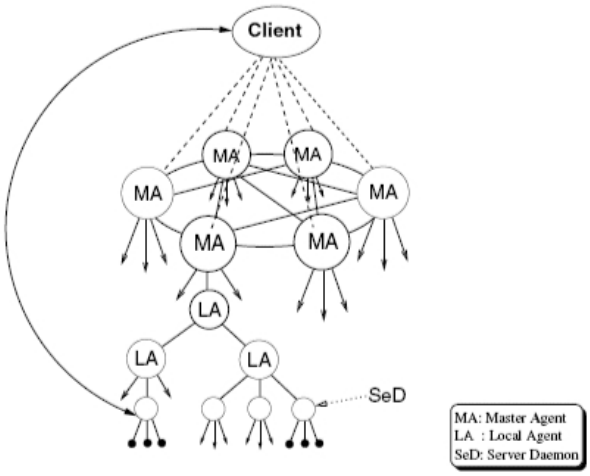

Fig. 1. The Diet distributed load balancer

In applications deployed over clusters (J2EE) or grids (DIET), the deployment of the servers is 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.

B. Component-based autonomic management

Component-based management aims at providing a uniform view of a software environment composed of different types of servers. Each managed server is encapsulated in 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. This solution is followed by several research projects, including Jade (Tune's predecessor) and Tune.

The component model we used in Tune is the Fractal component model [7]. A Fractal component is a run-time entity that is encapsulated and has one or more interfaces (access points to a component that supports a finite set of methods). Interfaces can be of two kinds: server interfaces, which correspond to access points accepting incoming method calls, and client interfaces, which correspond to access points

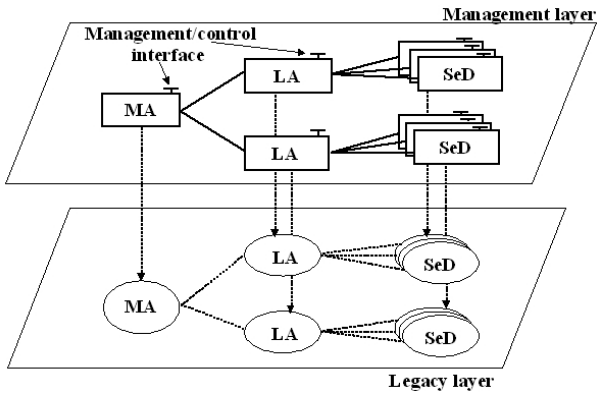

Fig. 2. Management layer for the Diet application

supporting outgoing method calls. The signatures of both kinds of interface can be described by a standard Java interface declaration, with an additional role indication (server or client). 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 Architecture Description Language (XML based language) 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. Any software managed with Tune is wrapped in a Fractal component which interfaces its administration procedures. Therefore, the Fractal component model is used to implement a management layer (Figure 2) 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 or the Master Agent in the case of Diet). 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, which are hidden in the wrappers.

Here, we distinguish two important roles:

- 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.
- 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. For instance, the configuration

of a LA server requires to know the name and location of the parent MA server it is bound to.

C. Motivations

Component-based autonomic computing has proved to be a very convenient approach. However, our experiments with full-scale applications revealed that:

- wrapping components are difficult to implement. The developper needs to have a good understanding of the component model which is use,
- deployment is not very easy. ADLs (Architecture Deployment language) 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 (as in the Diet example), deploying a thousand of servers requires an ADL deployment description file of several thousands of lines,
- 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 required a strong expertise regarding the used component model.

All these observations led us to the conclusion that a higher level interface was required for describing the encapsulation of software in components, the deployment of a software environment potentially in large scale and the reconfiguration policies to be applied autonomically. We present our proposal in the next section.

III. TUNE'S MANAGEMENT INTERFACE

As previously motivated, our goal is to provide a high level interface for the description of the application to wrap, deploy and reconfigure. An overall view of the management interface of Tune is proposed in Figure 3.

In our previous experiment with others system, we concluded that although there was a significant interest in relying on a component model such as Fractal for the implementation of a autonomic system, the interface provided by the component model we used was too low level. More precisely, in these systems, the administrator had to deal with the following tasks and associated tools:

- wrapping which had to be implemented using the programming interface of the Fractal component model. Our approach to this problem 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.
- deployment which was specified using Fractal's Architecture Description Language. Our approach to this issue is to introduce a UML-based metamodel for graphically describing deployment schemas. First, a UML based graphical description of such a schema is much more simple than an ADL specification, as it does'nt require expertise of the underlying component model. Second, the introduced deployment schema is more abstract than the previous ADL specification, as it describes the general organisation of the deployment (types of software to deploy, interconnection pattern) in intension, instead of describing in extension all the software instances that have to be deployed. This is particularly interesting for applications like Diet where thousands of servers have to be deployed.
- reconfiguration which had to be programmed imperatively in the Java language and using Fractal's APIs. Our approach to this issue is to introduce a UML-based metamodel for the description of state diagrams. These state diagrams are used to defined 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 an (concrete) architecture which conforms with the abstract schema, thus enforcing reconfiguration correctness.

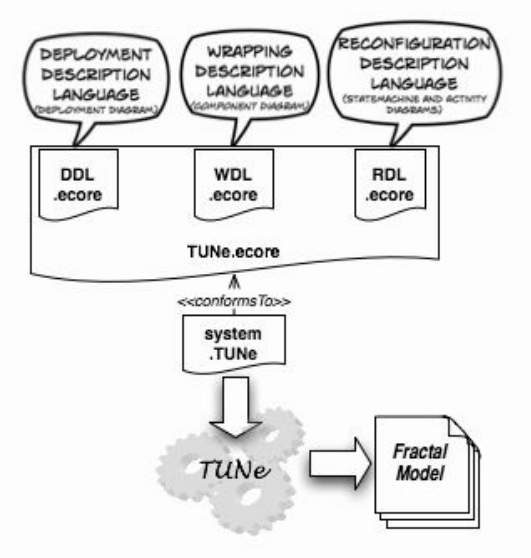

Fig. 3. Management interface of Tune

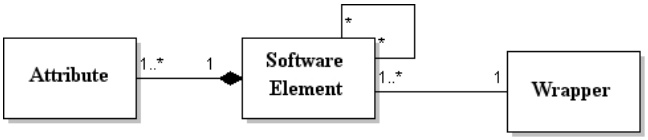

Fig. 4. A (conceptual) metamodel for deployment in Tune

We details each of these three aspects in the next subsections.

A. A metamodel for deployment schemas

We propose a UML-based metamodel to address legacy software deployment. A simplified version is shown in Fig-

ure 4. Each software to be deployed is represented by a *SoftwareElement*. Software elements can be interconnected by means of bindings. A software element is parameterized by a set of *Attribute* and it references its *Wrapper* (which interfaces the software). We now detail the semantics associated with this metamodel.

A deployment 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 software 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. An element includes a set of configuration attributes for the software (all of type String). Most of these attributes are specific to the software, but few attributes are predefined by Tune and used for deployment:

- **wrapper** is an attribute which gives the name of the WDL description of the wrapper,
- **legacyFile** is an attribute which gives the archive which contains the legacy software binaries and configuration files,
- **hostFamily** is an attribute which gives a hint regarding the dynamic allocation of the nodes where the software should be deployed,
- **initial** is an attribute which gives the number of instances which should be deployed. The default value is 1.

A use of our metamodel is illustrated by the model shown in Figure 5. It describes a Diet organization where one MA, two LAs and 10 SeDs (5 for each LA) should be deployed. A probe is linked with each software, which monitors the liveness of the server in order to trigger a repair procedure.

The schema in Figure 5 deploys a component architecture as illustrated in Figure 2.

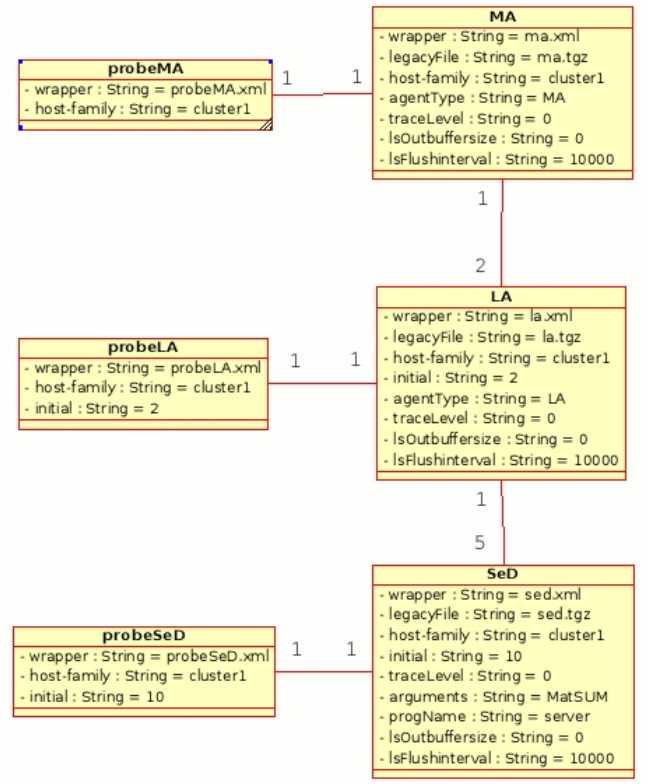

Fig. 5. Deployment schema for Diet

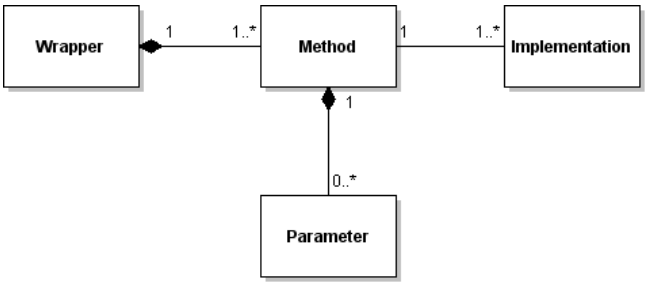

Fig. 6. Meta-model of the WDL

B. A Wrapping Description Language

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

The metamodel of the WDL language is shown in Figure 6. A wrapper is composed of a list of administration methods. Each method has an implementation and a list of effective parameters (values) that must be passed to the method upon invocation. We now detail the semantics associated with this metamodel.

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 order to configure and reconfigure the overall software environment is defined thanks to a dynamic diagram introduced in Section 3.3. Generally, a WDL specification 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 deployment 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:

- to hide the complexity of the underlying component model (Fractal),
- that most of the needs should be met with a finite set of generic methods (that can be therefore reused).

Figure 7 shows an example of WDL specification which wraps a SeD computing server in a Diet architecture. It defines *start* and *stop* methods which can be invoked to launch/stop the deployed SeD software, and a *configure* method which

reflects configuration attributes in the configuration file of the SeD software. The Java implementations of these methods are generic and have been used in the wrappers of most of the software we wrapped (LA, MA for Diet, but also Apache, Tomcat and MySQL for J2EE in an other case study. We only had to add an implementation of a configure method for XML configuration files). 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 deployment schema can be used to pass the configured attributes as parameters of the method invocations. However, several additional attributes are automatically added and managed by Tune:

- *dirLocal* is the directory where the software is actually deployed on the target machine,
- *compName* is a unique name associated with the deployed component instance,
- *PID* is the process identifier of the process that runs the software.

In Figure 7, the *start* method takes as parameters the shell command that launch the server, and the environment variables that should be set:

- *$dirLocal/$progName* is the name of the binary to be launched,
- *$dirLocal/$compName-cfg* is the name of the configuration file which is passed to the binary and which is generated by the configure method of the wrapper $arguments is a parameter for the binary,
- *LD_LIBRARY_PATH=$dirLocal* is an envionment variable to pass to the binary.

The *configure* method is implemented by the *ConfigurePlainText* Java class. This configuration method generates a configuration file composed of <attribute,value> pairs:

- *$dirLocal/$compName-cfg* is the name of the configuration file to generate,
- = is the separator between each attribute and value,
- and the attributes and value are separated by a ":" character.

It is sometimes necessary to navigate in the deployed component architecture in order to configure the software. For instance, in Diet, a LA has a configuration variable (in its configuration file) called *Name* which is a unique name associated with the launched server. This configuration variable is assigned with the *$compName* in its wrapper. A SeD which is a child of the LA must have a *parentName* configuration variable set to the name of the parent LA in its configuration file. Therefore, in the SeD wrapper (Figure 7), we need to access the *compName* of its parent LA in order to set this *parentName* configuration variable. Since in the deployment schema there is a link between the LA and SeD elements, there are bindings between the LA and the SeDs at the component level. These bindings allow navigating in the management layer. In Figure 7, the *parentName* configuration

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<wrapper name='sed'>
  <method name="start"
          key="appli.wrapper.util.GenericStart"
          method="start_with_pid_linux" >
    <param value="$dirLocal/$progName
                  $dirLocal/$compName-cfg $arguments"/>
    <param value="LD_LIBRARY_PATH=$dirLocal"/>
  </method>

  <method name="configure"
          key="appli.wrapper.util.ConfigurePlainText"
          method="configure">
    <param value="$dirLocal/$compName-cfg"/>
    <param value=" = "/>
    <param value="traceLevel:$traceLevel" />
    <param value="parentName:$LA.compName"/>
    <param value="name:$compName"/>
    <param value="lsOutbuffersize:$lsOutbuffersize"/>
    <param value="lsFlushinterval:$lsFlushinterval"/>
  </method>

  <method name="stop"
          key="appli.wrapper.util.GenericStop"
          method="stop_with_pid_linux" >
    <param value="$PID"/>
  </method>
</wrapper>
```

Fig. 7. A WDL specification for a SeD wrapper

variable is assigned with the name of the LA component which the SeD is bound with.

C. A metamodel for (re)configuration procedures

Reconfigurations are triggered by events. An event can be generated by a specific monitoring component (e.g. probes in the deployment schema) or by a wrapped legacy software which already includes its own monitoring functions. 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 Tune system 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 introduced a UML-based metamodel which allows specifying reconfiguration as state diagrams (we don't describe its meta-model here as it is very similar to that defined by UML). 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

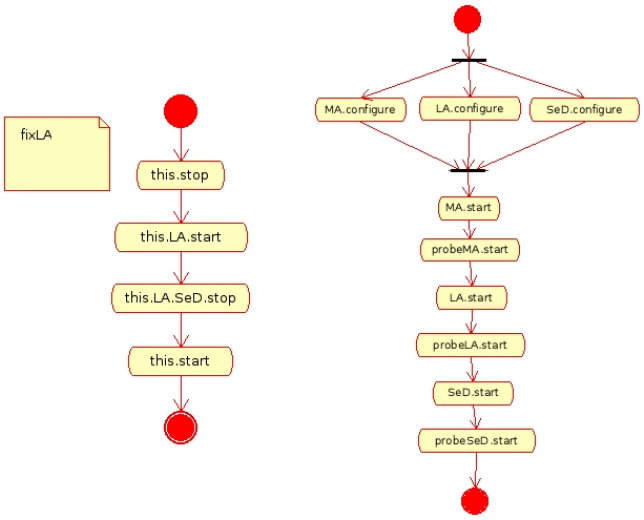

Fig. 8. Dynamic diagrams for repair and start in Diet

allows navigation in the component architecture, similarly to the wrapping language.

For example, let's consider the diagram in Figure 8 (on the left) which is the reaction to a LA (software) failure in Diet. The event (*fixLA*) is generated by a *probeLA* component instance, therefore this variable is the name of this *probeLA* component instance. Then:

- *this.stop* will invoke the *stop* method on the probing component (to prevent the generation of multiple events),
- *this.LA.start* will invoke the *start* method on the LA component instance which is linked with the probe. This is the actual repair of the faulting LA server,
- *this.LA.SeD.stop* will invoke the *stop* method on all the SeD component instances which are linked with this LA. This is necessary as in Diet, a restart of a LA requires to restart all its SeD children in order to reconnect to the LA. Here, the probes associated with the SeDs will trigger the restart of the SeDs,
- *this.start* will restart the probe associated with the LA.

Notice that state diagram's operations are expressed using the elements defined in the deployment schema, and are applied on the actually deployed component architecture. We are currently extending Tune to provide operations which re-deploy components (change location or add component instances) while enforcing the defined abstract deployment schema.

A similar diagram is used to start the deployed Diet environment, as illustrated in Figure 8 (on the right). In this diagram, when an expression starts with the name of an element in the deployment schema (LA or SeD ...), the semantic 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 MA, LA and SeDs. For each type of server, the server is started before its probe.

IV. Related Work

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 [8]. Management solutions for legacy systems are usually proposed as ad-hoc solutions that are tied to particular legacy system implementations (e.g. [9] 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 proposed by several projects [10][1][2][3] and has proved to be a very convenient approach, but in most of the 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.

Following the success of UML2 standard and the widespread of associated graphical editors, we propose in this paper a UML-based high level formalism which is composed of:

- a diagram for the description of wrappers,
- a diagram for specifying deployment schemas,
- a diagram for specifying reconfigurations as state transition charts.

Despite the introduction of many ADLs [11][12][13][14], we chose to define a UML2 subset which provides different diagrams to specify the different concerns of Tune. We rely on bindings between UML diagrams in order to enforce diagrams consistency. Another benefit of this approach is to allow reusing the numerous UML2 tools, including the UML2 TopCased editor[2] that we used in our project. Finally, this work led us to define a higher level formalisms for the administrator of a software environment and to formalize some UML variation points in the specific context of autonomic management.

V. Conclusion

Distributed software environments are increasingly complex and difficult to manage, and their administration consumes a lot of human ressources. To address this issue, many research projects proposed to implement administration as an autonomic software, and to rely on a component model to benefit from introspection and reconfiguration facilities that are inherent to these models. Although component-based autonomic computing has proved to be a very convenient approach, we observed that the interfaces of a component model are too low-level and difficult to use. In order to implement wrappers for legacy software, to describe deployed architectures and to implement reconfiguration programs, the

[2] http://www.topcased.org

administrator of the environment has to learn (yet) another framework with complex APIs or specific languages.

In this paper, we propose a higher level formalism for describing the encapsulation of software in components, the deployment of a software environment and the reconfiguration policies to be applied autonomically. This DSL for autonomic management is mainly based on UML for the description of deployment schemas and the description of reconfiguration state diagrams. A tool for the description of wrapper is also introduced to hide the details of the underlying component model. We are currently extending our prototype to enable various form of reconfigurations (not just invocations on wrapper's methods). Notably, we provide support in state diagrams to allow component re-deployment, i.e. changing a component's location and adding a component instance, while still conforming to the specified abstract deployment schema.

ACKNOWLEDGMENT

The work reported in this paper benefited from the support of the French National Research Agency through projects Selfware (ANR-05-RNTL-01803), Scorware (ANR-06-TLOG-017) and Lego (ANR-CICG05-11).

REFERENCES

[1] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self adaptation with reusable infrastructure," in *IEEE Computer, 37(10)*, 2004.

[2] D. Hagimont, S. Bouchenak, N. D. Palma, and C. Taton, "Autonomic management of clustered applications," in *IEEE International Conference on Cluster Computing, Barcelona*, September 2006.

[3] P. Oriezy, M. Gorlick, R. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A.Wolf, "An architecture-based approach to self-adaptive software," in *IEEE Intelligent Systems 14(3)*, 1999.

[4] *Unified Modeling Language (UML) 2.1.1 Superstructure Specification*, Object Management Group, Inc., july 2005, final Adopted Specification. [Online]. Available: http://www.omg.org/docs/formal/07-02-05.pdf

[5] *Unified Modeling Language (UML) 2.0 Infrastructure Specification*, Object Management Group, Inc., aug 2003, final Adopted Specification. [Online]. Available: http://www.omg.org/docs/ptc/03-09-15.pdf

[6] P. Combes, F. Lombard, M. Quinson, and F. Suter, "A scalable approach to network enabled servers," in *In 7th Asian Computing Science Conference*, January 2002.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani, "The fractal component model and its support in java," in *Software - Practice and Experience, special issue on "Experiences with Auto-adaptive and Reconfigurable Systems", 36(11-12):1257-1284*, September 2006.

[8] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," in *IEEE Computer Magazine, 36(1)*, 2003.

[9] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisiong of multi-tier internet applications," in *2nd International Conference on Autonomic Computing (ICAC-2005), Seattle, WA*, June 2005.

[10] G. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas, "Reflection, self-awareness and sef-healing in openorb," in *1st Workshop on Self-Healing Systems, WOSS 2002*, 2002.

[11] J. Magee and J. Kramer, "Dynamic structure in software architectures," in *4th ACM SIGSOFT symposium on Foundations of software engineering*, October 2002.

[12] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proceedings of CASCON'97*, November 1997.

[13] J. V. D.C. Luckham, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, September 1995.

[14] N. Medvidovic, D. Rosenblum, and R. Taylor, "A language and environment for architecture-based software development and evolution," in *21st international conference on Software engineering*, May 1999.