Metamodeling Autonomic System Management Policies

Ongoing Works

Benoît Combemale, Laurent Broto, Alain Tchana and Daniel Hagimont

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

1 Introduction

Autonomic computing is recognized as one of the most promising solution to address the increasingly complex task of distributed environments' administration. In this context, many projects relied on software components and architectures to organize such an autonomic management software.

However, we observed that the interfaces of a component model are too low-level, difficult to use and still error prone. Therefore, we introduced higher-level languages for the modeling of deployment and management policies. These domain specific languages enhance simplicity and consistency of the policies. Our current work is to formally describe the metamodels and the semantics associated with these languages.

After an introduction of the research context and our motivations in Section 2, we present our approach to the modeling of management policies. We first overview in Section 3 the UML-based language support we provide for management policy specification. We then describe in Section 4 the metamodels associated with these languages. We conclude this position paper in Section 5 with the perspectives that this work opens.

2 Background

2.1 Autonomic Computing

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.

2.2 Component-based Autonomic Systems

Many works in this area have relied on a component model to provide such an autonomic system support [4, 6, 11]. 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, configuration, deployment and reconfiguration facilities, in order to implement their autonomic management processes.

In a previous project, we designed and implemented such a component-based autonomic management system (Jade [6]). In the Jade system, an administra-

[0]This work is supported by the RNTL project SCORWARE (contract ANR-06-TLOG-017), cf. http://www.scorware.org.

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

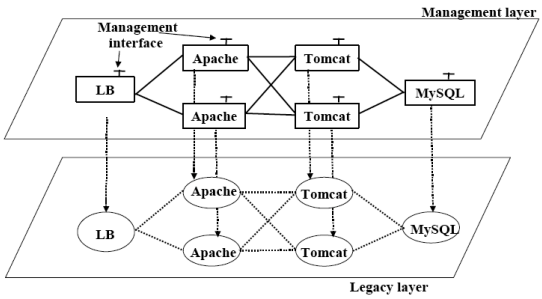

Figure 1. Management layer

tor can wrap legacy software in components (Jade relies on the Fractal component model [2]), describe a software configuration to deploy using the component model ADL (*Architecture Description Language*) and implement reconfiguration programs (autonomic managers) using the component model's interfaces (Java interfaces in Fractal).

Therefore, the Fractal component model is used to implement a management layer on top of the legacy layer (composed of the actual managed software). Figure 1 illustrates this management layer for a classical J2EE software architecture where several middleware tiers (request load-balancer, Apache, Tomcat, MySQL) are combined and interconnected.

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. 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.

Component-based autonomic computing has proved to be a very convenient approach. The experiments we conducted with Jade [6] for managing cluster or grid infrastructures validated this design choice. But as Jade was used by external users (external to our group), we observed that:

- wrapping components are difficult to implement. The developper needs to have a good understanding of the component model we use (Fractal).
- configuring an architecture to be deployed is not very easy. 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, deploying a thousand of servers requires an ADL 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 (Fractal's APIs) of the management layer. This also required a strong expertise regarding the used component model. Moreover, it is difficult to enforce consistency of the managed infrastructure whenever it is reconfigured.

3 Policies specification in Tune

All the previous observations led us to the conclusion that higher-level support was required for describing the encapsulation of software in components, the configuration and deployment of a software environment potentially in large scale and the reconfiguration policies to be applied autonomically. Tune is an evolution of Jade which aims at providing a higher level formalism for all these tasks (wrapping, configuration, deployment, reconfiguration). The main motivation is to hide the details of the component model we rely on and to provide a more intuitive policy specification interface. A more detailed description of management policy specification in Tune is available in [1].

3.1 A language dedicated to wrapping description

Regarding wrapping, our approach is to introduce an XML-based 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. She only has to describe the outline of the wrapper, including references to some Java packages which provide reusable

basic functions for configuring the wrapped legacy software (generally configuration files).

3.2 A language dedicated to architecture description

Regarding the description of the architecture to be deployed and administrated, our approach is to use a graphical language for describing architectures. First, a UML-based graphical description of such an architecture is much more intuitive than an ADL specification, as it does'nt require expertise of the underlying component model. Second, the described architecture is more abstract than the previous ADL specification, as it describes the general organisation of the architecture (types of software, interconnection pattern) in intension, instead of describing in extension all the software instances that compose the architecture. This is particularly interesting for grid applications where thousands of servers may compose an architecture to be deployed.

3.3 A language dedicated to deployment description

An architecture can be projected on an abstraction of the deployment environment. The environment abstraction is composed of abstract nodes. An abstract node may correspond to a single machine or a group of machines, in which case an allocation policy is associated with this group.

The projection of an architecture describes the number of instances that should be deployed in each abstract node.

Regarding this aspect, we provide a language which allows the description of a deployment environment and the projection of a software architecture in this environment.

3.4 A language dedicated to reconfiguration description

Regarding reconfiguration, our approach is to use the UML graphical language for the description of state diagrams. These state diagrams are used to define workflows of operations that have to be performed for reconfiguring the managed environment. One of the main advantage of the introduced language, besides simplicity, is that state diagrams manipulate the entities described in the architectural schema (previous subsection) and reconfigurations can only produce an (concrete) architecture which comforms with the architectural schema, thus enforcing reconfiguration correctness.

4 Tune Metamodeling

Our first experiments with Tune focussed on the use of XML and UML to take advantage of many existing open source tools. We used the graphical editors provided by the TOPCASED Eclipse-based toolkit [3] for the description of architectures and reconfiguration diagrams.

However, the use of this unified language led us to specialize (pragmatically but sometimes awckwardly) its initial semantics in order to adapt it for the autonomic computing field. 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 autonomic system management policies definition. This would allow us to decline textual or graphical editing tools, offering a constraint, domain-specific and user-friendly formalism. Although this work is still in progress, we detail in this section the followed approach and the promising results that we obtained. We present in a first step the various points of view that we have taken into account in defining an autonomic system like Tune. We also introduce each corresponding metamodel. We conclude this section by a presentation of the first editing prototypes that we have already defined.

4.1 Concerns about Tune Systems

As part of our initial work to define a modeling language for system management policies, we have taken into account four main concerns (fig. 2). We describe each of them in the remainder of this section.

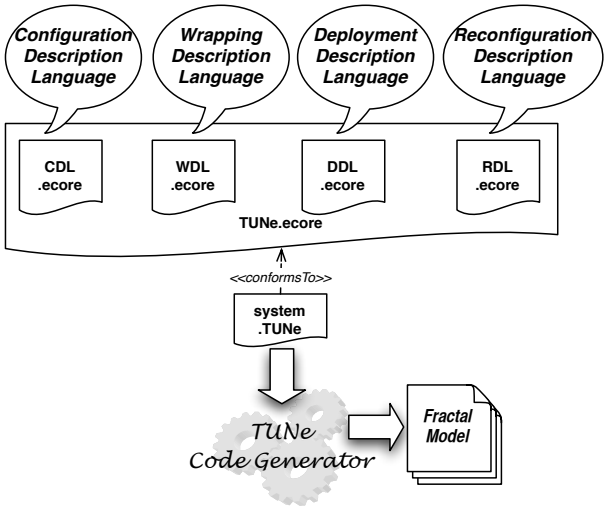

Figure 2. Concerns in Tune

4.1.1 The Configuration Description Language

The first one is the homogeneous definition of architecture of the application. We propose 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 3.

The main concept of this view is the *SoftwareElement* describing a particular type of software with its own configuration, management, and independent life cycle. 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 described by different *Softwa-*

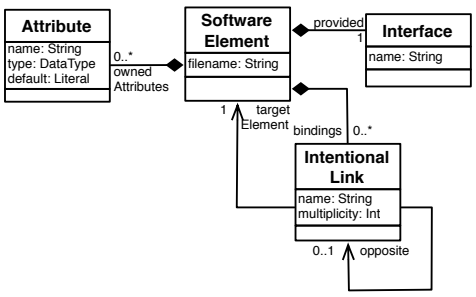

Figure 3. The (simplified) CDL

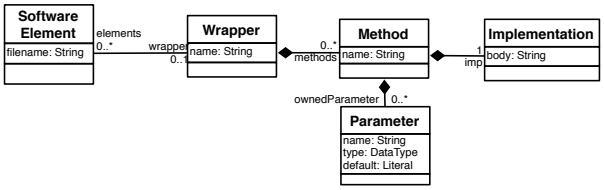

Figure 4. The (simplified) WDL

reElements 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 *bindings*, allowing to connect a *SoftwareElement* to another, and expressing a *multiplicity* and a role (*name*). The multiplicity expresses the number of instances of the target *SoftwareElement* for each one of the source *SoftwareElement*. The role allows navigation with a query language relying on OCL [9]. It is also possible to define bi-directional *bindings* by defining an *opposite bindings*.

4.1.2 The Wrapping Description Language

The second concern is the definition of a wrapper and its relation with *SoftwareElements*. The corresponding metamodel is presented in Figure 4.

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 *SoftwareElements*) 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 the OCL constraints to verify that the wrapper associated with a software element defines at least the methods provided by the interface.

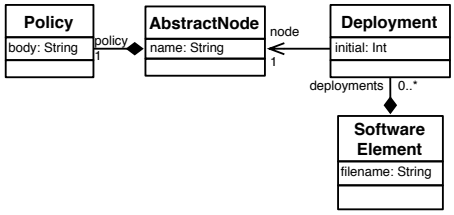

Figure 5. The (simplified) DDL

4.1.3 The Deployment Description Language

The third view in Tune, described by the metamodel presented in Figure 5, is used to define by intention or by extension, the real deployment of instances of each software component on system's nodes. 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.

4.1.4 The Reconfiguration Description Language

The last view makes possible to describe the life cycle of a software element with schemas of automatic configuration and reconfiguration. The described policies are triggered by endogenous (i.e. from the legacy level) or exogenous (sent by the administrator or from a workload) events defined from triggers (*Trigger*) for each transition.

To describe this life cycle, we rely on state and activity diagrams of UML2.0 [10, §12–§15]. A simplified representation is presented in Figure 6. A transition triggers the execution of an activity diagram (*AutonomicConfiguration*) whose actions are a partially-ordonned sequence of wrapper's method calls.

4.2 Domain Specific Tools

One of the main motivations to have defined an abstract syntax dedicated to Tune was to be able to provide domain-specific and user-friendly tools. For this purpose, we use the set of generative tools provided by the community of the MDE. This allows to generate editors, with textual (e.g. TCS [7] or Syntaks [8]) or graphical (e.g. Topcased [3] or GMF [5]) formalisms.

In Tune, we have currently defined a textual editor for the WDL (a screenshot of the first prototype is depicted in Figure 7).

5 Conclusion & Perspectives

We are investigating the design and implementation of an autonomic system called Tune. Tune relies on a component model in order to provide support for encapsulating (wrapping) software, describing the software architecture to manage and its deployment in 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 the administrator in its policy description tasks.

For this purpose, our first experiments focused on the use of the UML formalism (and numerous tools that supports it, e.g. TOPCASED). These experiments confirmed the interest of raising the abstraction level but we had to specialize the UML semantics according to the requirements of the considered field (autonomic

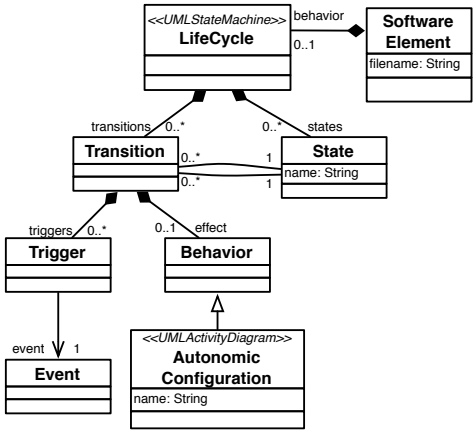

Figure 6. The (simplified) RDL

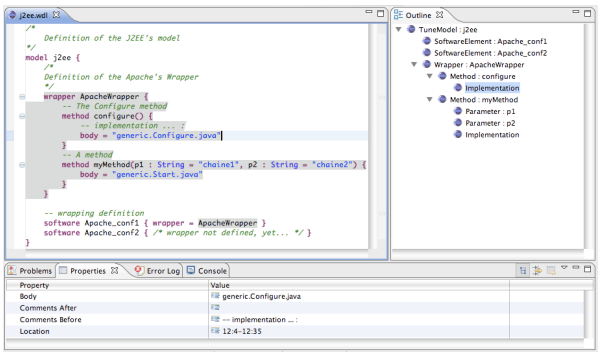

Figure 7. Clustered J2EE servers with MDE4Tune: the WDL textual editor

administration and the expression of Tune's policies). It was difficult to take into account this specialization in the tools we reused.

We are currently working on a dedicated metamodel. The expected benefits are two-fold: to provide a formal definition of Tune's policy description languages, to statically and dynamically validate the policies described by the administrators with customized editors.

This position paper opens many perspectives on which we are currently working. In the short term, we hope to finalize the Tune metamodel and provide some editing tools for the different points of view. The Tune core team is currently extending the reconfiguration capabilities of Tune and the metamodel and the editing tools are evolving 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 Management*. 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] L. Broto, D. Hagimont, P. Stolf, N. Depalma, and S. Temate. Autonomic management policy specification in Tune. In *23rd Annual ACM Symposium on Applied Computing, Fortaleza, Brésil*, March 2008.

[2] 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.

[3] P. Farail, P. Gaufillet, A. Canals, C. L. Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic Systems Design. In *Embedded Real Time Software*, Toulouse, Jan. 2006.

[4] 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.

[5] GMF. Graphical Modeling Framework. `http://www.eclipse.org/gmf/`.

[6] 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.

[7] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *5th International Conference on Generative Programming and Component Engineering*, pages 249–254, Portland, Oregon, USA, Oct. 2006. ACM.

[8] P.-A. Muller, F. Fleurey, F. Fondement, M. Hassenforder, R. Schneckenburger, S. Gérard, and J.-M. Jézéquel. Model-Driven Analysis and Synthesis of Concrete Syntax. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *9th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 98–110, Genova, Italy, Oct. 2006. Springer.

[9] Object Management Group, Inc. *UML Object Constraint Language (OCL) 2.0 Specification*, June 2005.

[10] Object Management Group, Inc. *Unified Modeling Language (UML) 2.1.1 Superstructure*, Feb. 2007. Final Adopted Specification.

[11] 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. *IEEE Intelligent Systems 14(3)*, 1999.