Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-based Systems

Jérémy Dubus, Philippe Merle

INRIA-Futurs Projet Jacquard / LIFL
Université des Sciences et
Technologies de Lille (USTL)
59655 Villeneuve d'Ascq, France
{Jeremy.Dubus,Philippe.Merle}@inria.fr

ABSTRACT

Manual administration of complex distributed applications is almost impossible to achieve. On one side, work in autonomic computing focuses on systems that are able to maintain themselves, driven by high-level policies. Such a self-administration relies on the concept of a control loop. On the other side, modeling is currently used to ease design of complex distributed systems. Nevertheless, at runtime, models remain useless, because they are decoupled from the running system which is subject to dynamic changes. The autonomic computing control loop involves an abstract representation of the system used to analyze the situation and to adapt the application properly. Our proposition, named *Distributed Autonomous Component-based ARchitectures* (DACAR), introduces models in the control loop. Using adequate models into the control loop, it is possible to design both the distributed systems and their evolution policies, and to execute them. The metamodel suggested in our work mixes both OMG Deployment and Configuration specification and the Event-Condition-Action (ECA) metamodels. This paper treats the different concerns that are present in the control loop and focuses on the concepts of the metamodel that are needed to express entities of the control loop. It also gives an overview of the current DACAR prototype and illustrated it on an ubiquitous application example.

Keywords

Autonomic computing, Metamodeling, Distributed systems, OMG D&C, ECA rules, Dacar

1. INTRODUCTION

Business applications become more and more complex, and they are also distributed on several machines. The resulting heterogeneity of the *deployment domain* —*i.e.* the set of machines that host these applications— makes deployment and maintenance of these applications become critical tasks. Actually with the emergence of grid and ubiquitous computing [5, 18], manual administration of applications is almost impossible to achieve, since the deployment domain is not statically known at deployment-time, and it can strongly evolve during runtime, in terms of node appearance or disappearance. This statement led to the creation of a new research topic called *Autonomic Computing* [13]. Work in autonomic computing focuses on systems that are able to maintain themselves during runtime, driven by high-level policies. In autonomic computing, the *control loop* is the central notion that helps achieving autonomic management and reconfiguration of applications. This control loop involves an abstract representation of the system used to analyze the situation and to adapt the application properly. There is a causal link between the abstract representation of the system and the actual system that is running.

Models are widely used to design distributed applications. The use of models allows the designer to only specify an abstract view of a system to be deployed. Using this abstract models, many approaches, such as the *Model Driven Architecture* [8], are able to generate a more concrete view of the system, and the task of the designer is drastically simplified. Unfortunately, the models of an application are useless once the system is deployed and running, since models and system are decoupled and they change independently. Introducing models as the abstract representation of a system in the autonomic computing control loop makes the use of models still relevant at runtime. The model is causally linked to the running system, and it evolves the same way. Then the autonomic computing can benefit from the use of models by having a complete abstract representation of the system from which it is possible to extract fine-grained information, and then apply the adequate reconfiguration.

DACAR is our proposition that consists in establishing a metamodel that mixes the OMG Deployment and Configuration (D&C) specification [10] and Event-Condition-Action (ECA) rules [3], in order to inject models as computation support in autonomic computing.

The remainder of the paper is organized as follows. Section 2 presents the fundamental principles of autonomic computing as defined in [13]. Section 3 discusses the key research challenges to address the problem of introducing models in autonomic computing. Section 4 presents DACAR, our proposition for model-based autonomic computing. A simple illustrative example is given in Section 5. Section 6 presents other work related to autonomic distributed component-based systems. Finally, Section 7 exposes our conclusions and perspectives.

2. PRINCIPLES OF AUTONOMIC COMPUTING

The essence of autonomic systems relies on the notion of *control loop* (represented in Figure 1) as defined in [13]. This loop consists in four phases: *Monitoring* the system, *analyzing* the situation to take a decision about some changes monitored, *planning* the adequate reconfiguration actions, and *applying* them. The analyzing and planning phases are relying on the *Knowledge* part as support for computation. At runtime the Knowledge part must always be conform to the execution environment, that means that every change in the execution environment must lead to an update of the knowledge part, and *vice-versa*. Then a *causal link* must be maintained between the knowledge part and the execution environment.

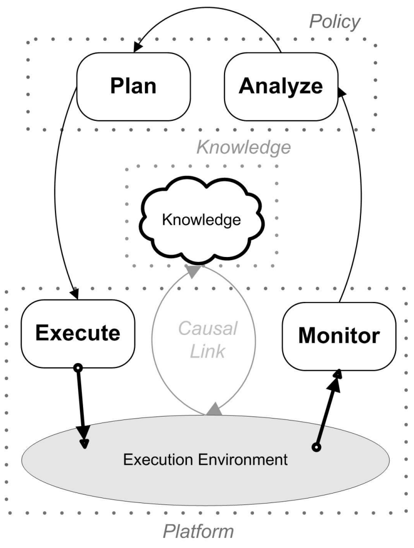

Figure 1: The control loop of autonomic computing

We choose to separate the architecture of the control loop into three parts which have different roles:

The knowledge part It is the abstract autonomous system representation. This representation must be complete in order to be aware of any information that may influence decision to take. It must be reconfigurable, because the abstract representation must evolve the same way that the running application onto the deployment domain. It must also be an high-level representation where only relevant information about the application must be present, so that the autonomic mechanism does not get lost into details when computing a decision.

The policy part The *Analyze* and *Plan* phases are responsible of exploiting the knowledge part to analyze situation and prepare adequate reconfigurations. We group them into the policy part. The integrality of the autonomic policy is gathered in these two phases.

The platform part The third part encompasses the execution environment, and the *Monitor* and *Execute* parts. These three parts represent the middleware used. It encompasses operations to deploy the system, reconfigure it, and monitor it. Those operations are dependant of the underlying software technology used.

3. KEY RESEARCH CHALLENGES

This section discusses the key research challenges to address for introducing models into autonomic computing. Then we try to answer to the following questions: What models of autonomic applications should look like? How the control loop could be split? What metamodels can be used to provide concepts to express entities of the control loop? Section 2 already provided answer's elements. The control loop architecture has been split in three parts that represent the different concerns of an autonomic system.

First, the platform part of the control loop should be modeled. We saw that this platform should provide monitoring information about the running application as well as deployment operations to instantiate the application onto the deployment domain. Then, there is no possibility of a generic complex metamodel in this case, since a unique metamodel could hardly be used to express any runtime platform model. We can then suppose that the platform part is encapsulated in one software component with clearly defined interfaces providing the required operations. The details about the implementation of this component relies on the runtime platform chosen for executing the application. By using a platform component, we maintain genericity regardless to middleware running the applications (*e.g.* J2EE [15], CCM [9], SCA [12], etc.).

The second part is about the Knowledge part. What information about the application a knowledge model should encompass? This part must contain all information relevant to analyze the system and to plan adequate reconfiguration actions. This model must provide the concepts to:

- Describe deployment domain entities: Computers available on the network, interconnections between those computers, bandwidth of these interconnections, etc. These concepts are needed since in the case of an ubiquitous or grid-tailored application, the knowledge about computer's appearance/disappearance is critical.
- Describe the components of the applications. This encompasses the component types, the interfaces they provide and require, the location of the binary of these components, etc. This information is needed because the reconfiguration of running component instances requires knowledge of their specification.
- Express the structure of the concerned applications, which we also call the *Deployment Plan*. This means information about the component instances to deploy, the computers onto which these instances must be deployed, and how these instances are bound together. This information is crucial for the application since it contains all information about the application architecture. Without knowing this information it is im-

possible to deploy and start the application. Moreover autonomic management of applications often means reconfigurate dynamically the structure of an application, so it means modifying the deployment plan.

There are already existing solutions, like the *Architecture Description Languages* (ADL) [14], in order to modelize application data such as component types, deployment plan, etc.. But most of the existing ADLs are specific to a given component model and the few that are generic generally do not provide concepts to express deployment domain entities. Nevertheless the Object Management Group (OMG) has recently adopted a new specification called *Deployment and Configuration of Distributed Component-based Applications* [10]. This specification defines a metamodel with two parts, the *Component Data* part and the *Component Management* part. The first one describes the packaged components, with their typed interfaces and implementations, whereas the second one describes the deployment infrastructure and the way it handles data from the first part to execute the deployment process. The Component Data part of the OMG D&C specification provides the three parts of our knowledge metamodel: the *Types*, the *Domain* and the *DeploymentPlan*. So it seems that OMG D&C is a convenient knowledge metamodel to use.

However, using OMG D&C means forgetting some of the aspects of a running system. For instance, it is impossible to express components container policies, such as transactions, security, persistency or lifecycle concerns. Nevertheless, there exist extensions to specify extra-functionnal container policies, such as the CORBA Component Descriptors (CCD). So in order to reify and control every fine-grained concern of an component-based application, the OMG D&C metamodel should be extended.

The third part is about the high-level autonomic policy. This part must describe exhaustively the autonomic policies of an application. How should these policies be expressed? Which concepts can allow the designer to specify precisely the exhaustive autonomic policies of a distributed component-based application, knowing that the model at runtime should have sufficient information to analyze the situation and to take the right decision when needed?

There are three main concepts in order to express an autonomic policy. First, there is a *stimulus* part, that is an event that triggers the autonomic policy. This stimulus emerges in a certain *context*, which can be modeled as a set of properties of the application or the stimulus itself. Finally, there is an *execution* part that depends on the nature of the stimulus as well as the context in which this stimulus is detected. We argue that the ECA rule paradigm, well-known in the domain of active databases [3], fits well our needs for the expression of autonomic policies. The Event part represents the stimulus, the conditions of a rule are the context of this rule trigger, and the actions represent the execution part of the model.

Another issue to be raised about this part of the model is to know whether the policy expression must be fine or coarse-grained. In other words, the question is about having an application-specific autonomic policy or composing generic fine-grained policies to build an application global policy. Our opinion is that by composing fine-grained policies, it will be possible to extract independent and reusable autonomic micro-policies. It is then possible to define policies by composing both independant and more application-specific policies. On the other way the well-known feature interactions defined in [17] can be detected and resolved thanks to the fine granularity of our rules.

The *Shared Trigger Interaction* (STI) problem occurs when two or more rules are triggered at the same time (*i.e.* by the same event). In our situation we consider that no rule of the same category should be triggered by the same event. Then there can be two alternatives to solve this problem. The first one is simplistic: Dialog with administrator to choose whether these rules can be merged or executed in any order, choose an execution order, or remove some of the rules because they have incompatible actions. The second alternative is more complex. We can add in the metamodel post-conditions properties for each rule (which can be inferred from the rule code). It is then possible to detect if two or more rules that share triggers have compatible post-conditions (that is to say different but compatible behaviours), or if these rules are also sharing the target of the reconfiguration. In this latter case, there is no other choice than prompting the administrator.

The *Sequential Action Interaction* (SAI) problem occurs when the action of a rule leads to the triggering of another rule. But in our specific case, this is not really a problem. But if these SAIs lead to a *Looping Interaction* (LI), then it has to be resolved. LI occurs when there is a cycle in the rules triggering. The typical simple illustration is the action of a rule `A` that triggers a rule `B`, and the action of `B` triggers `A`. Using the post-condition mechanism exposed earlier it is firstly possible to detect SAIs when inserting a new rule, by comparing events that triggers a rule with post-condition of another rule. This way an algorithm for detecting cycle is envisageable, and it is then possible to rollback the rule insertion.

To experiment the mixing of OMG D&C metamodel and ECA rules for the control loop of autonomic applications, we have created a prototype that allows us to design, deploy, and reconfigure autonomic distributed component-based applications.

4. OUR DACAR PROTOTYPE

DACAR for *Distributed Autonomous Component-based ARchitectures* is our prototype to build autonomic distributed CORBA component-based applications. The platform employed to deploy components is OpenCCM (`http://openccm.objectweb.org`). We describe applications using XML descriptors conform to OMG D&C. These descriptors are reified into memory as a graph of Java objects that represents the executable model of the running applications. Currently, rules are not expressed using a model since our rule metamodel has to be completely defined. So the rules are implemented as Fractal lightweight components (`http://fractal.objectweb.org`). Monitoring and reconfiguration execution parts are implemented using specific OpenCCM mechanisms [11].

We consider two sorts of events: The *endogenous* events are coming from the knowledge part (*example:* A new instance description has been added in the domain part), the *exogenous* events are events coming from the execution platform (*example:* A new node has been started in the deployment domain). The *condition* part of a rule represents conditions that event properties must fulfill in order to trigger the rule. The *action* part can have effect either on the knowledge part or the platform. Thus, we can classify rules in three categories (represented on Figure 2):

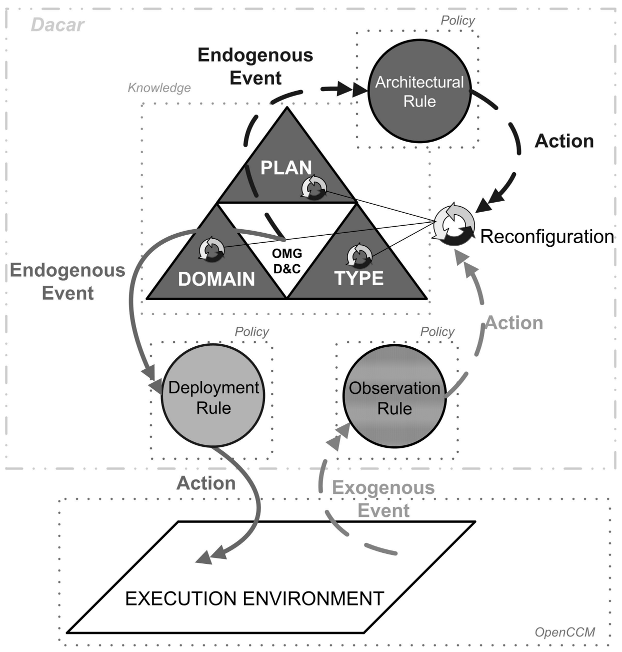

Figure 2: The three types of rules involved in Dacar platform

- The *Monitoring rules* are triggered by exogenous events. They operate actions on the knowledge part to update it according to changes that occurred onto the running platform. *Example:* When a new node is detected in the execution environment, add its description in the domain part of the knowledge part. These rules are generic and reusable across applications.
- The *Deployment rules* are triggered by endogenous events. They operate actions on the running platform to update it according to changes that occurred onto the knowledge part. *Example:* When a new instance is declared in the deployment plan part of the knowledge part, prepare the deployment of this instance on the execution environment. These rules are also generic and reusable across applications.
- The *Architectural rules* are triggered by endogenous events. They operate actions on the knowledge part to update it according to properties that this knowledge part must fulfill. *Example:* When a new `Client` instance is declared in the plan of the knowledge part, declare a binding between this `Client` component and a random `Server` component instance existing in the plan. These rules are application specific.

Using these three categories of rules it is possible to ensure the causal link between the knowledge part and the application at runtime. The monitoring rules ensure that every change occuring in the execution environment leads to an update operation of the knowledge part. The deployment rules ensure that every concept declared in the plan of the knowledge part is prepared to be deployed on the execution environment. Both monitoring and deployment rules are the generic micro-policies enounced in Section 3. Finally, the architectural rules are the only specific part of the policy, they are responsible of the adaptation policy of the application. It refines the knowledge part according to changes emerging from the knowledge part itself. It is possible to express complete autonomic applications using our metamodel. DACAR has been tested to design several autonomic CORBA component-based applications.

5. SIMPLE EXAMPLE

This section illustrates the DACAR concepts through a simple scenario of autonomous application. Figure 3 represents the architecture of our scenario. More details about this scenario can be found in [4].

This example takes place in the context of an ubiquitous application. In a railway station, there is a `RailwayStation` component that can give information about the trains on departure, relying on a `DataBaseTrainSchedule` component. Every person that enters the station and has a Personal Digital Assistant (PDA) must be able to request the `RailwayStation` component. In order to realize this, a dedicated `TrainGUI` component is implemented and must be deployed on every PDA that wants to obtain the service. With most of existing ADLs, it is impossible to specify that a `TrainGUI` component must be deployed on every PDA that enters the domain. Moreover, those deployed `TrainGUI` components must be bound to the `RailwayStation` instance.

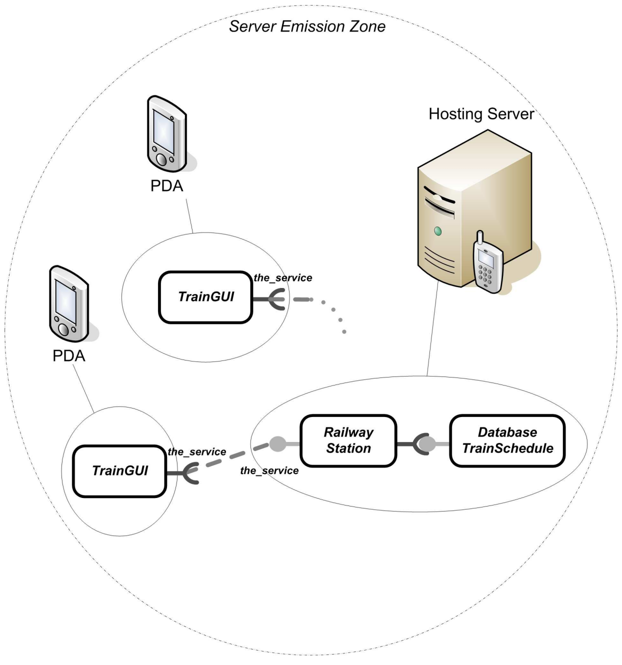

Figure 3: An autonomic train service example

We can first introduce the generic monitoring rule `R_M`, in charge of adding new node descriptions into the domain part of the autonomic computing knowledge:

```
RULE R_M
EVENT
    A new node N is detected onto the Platform
CONDITION
    N.profile == PDA
ACTION
    knowledge.domain.addNode(N)
```

Two architectural rules are required to implement autonomic behaviours of our simple ubiquitous example. The first one is the following :

```
RULE R1
EVENT
    A new node N is declared in the knowledge.domain part
CONDITION
    N.profile == PDA
ACTION
    knowledge.plan.declareInstance(TrainGUI ,
    "cl"+N.name, N)
```

This rule `R1` ensures that every terminal that enters the domain gets an instance of `TrainGUI`. The second rule ensures that every `TrainGUI` is connected to the `RailwayStation` component :

```
RULE R2
EVENT
    New instance I is declared in the knowledge.plan part
CONDITION
    I.type == TrainGUI
ACTION
    knowledge.plan.addConnection(I.the_service,
    RS.the_service)
```

With these only two rules, the architecture will be extended to take into account every PDA that enters the domain. These rules reuse generic deployment and monitoring rules that are respectively in charge of applying the deployment operations when needed, and adding the description of new nodes into the reified architecture when it is detected. The unbinding and undeployment of `TrainGUI` instances when PDAs leave the domain must be written (also two rules) but is not treated here.

We can also give details about one generic deployment rule `R_D`, that is in charge of deploying component instances declared in the plan:

```
RULE R_D
EVENT
    A new instance I is declared in the knowledge.plan part
CONDITION
    true (no condition)
ACTION
    platform.deployInstance(I)
```

The potential gains of the approach we present in this paper are numerous. First, the ECA rules represent a convenient and natural way to express reconfiguration policies, as well deployment and monitoring operations. Also, architectural rules can be designed to factorize the description of very large and redundant applications, just like the simple example we gave in this section. Indeed in our example, the same actions are repeated when new PDA enter the domain, but the two concise architectural rules factorize these actions Finally for critical systems, there is no need to monitor the application and to manually interact with the system whenever an human intervention — that is also error-prone— is needed. An arbitrary part of the application can be self-managed at runtime.

6. RELATED WORK

In this section, we will discuss of works addressing distributed component-based autonomic applications, in order to justify the relevance of the metamodels we defined and the way we interpret models at runtime to deploy and execute autonomic applications.

JADE proposes a component-based implementation of a control loop to administrate J2EE applications on clusters [2]. The target platform and the application are modeled using Fractal components, in order to provide management interfaces. This allows the administrator to dynamically reconfigure the application architecture. A *sensor* mechanism is employed to monitor the system and communicate the observations to the control loop. JADE allows the architectures to be reconfigured according to infrastructure context changes. It does not provide way to express architecture-specific adaptation mechanisms. Moreover the knowledge part of JADE consists only in a Fractal component assembly which is not as expressive as a real typed model defined by a metamodel.

CoSMIC is a model-driven generative programming toolchain, that permits efficient deployment and reconfiguration in Distributed Real-time Embedded (DRE) systems [7]. It is also based on the OMG D&C specification to specify deployment process. Nevertheless, CoSMIC only monitors Quality-of-Service results. Autonomous reconfigurations are then triggered by performance leaks in the system. This adaptation process is not driven neither by the architecture nor by its deployment domain evolutions, by opposition with our approach.

The RAINBOW framework adopts an architecture-based approach very similar to ours, to adapt distributed applications to their needs [6]. They also implement a control loop to manage elements across the systems. They define *adaptation strategies* using *invariants*, which are some reconfiguration scripts executed in response to events. The use of invariants makes the policies in RAINBOW monolithic, on the contrary of our approach. This way RAINBOW's invariants interactions must be hard to detect and hard to resolve. Using our rule-mechanism, it is possible to detect interactions between policies. Then, autonomic reconfigurations could be validated and safe at runtime.

Finally, PLASTIK is a meta-framework that provides mechanisms to manage runtime reconfigurations of component-based software, with *programmed changes* —*i.e.* foreseen reconfigurations at design-time— versus *ad hoc changes* —*i.e.* not foreseen at design-time [1]. This approach relies on reactive reconfigurations, in the same way that our approach. It proposes two layers, an architectural one, and a runtime one, just like in DACAR. Nevertheless, the only coherency supported between the two layers is from the ADL layer to the platform layer. This means that in case of runtime-level spontaneous changes, the architectural representation of the system is deprecated, and then useless. In our approach, a causal link is maintained between the two layers.

7. CONCLUSIONS & FUTURE WORK

In this paper, we have presented our vision of model-based autonomic computing in which a metamodel composed of three parts have been found out to express distributed component-based autonomic applications. This led to the implementation of DACAR, a framework to model autonomic component assemblies, following the vision of autonomic computing. DACAR reuses the OMG D&C architecture metamodel to build and manipulate the knowledge part of the control loop, and reuses the ECA rules paradigm to express the applicative adaptation policies. The first point of our future work will consist in establishing precisely the exhaustive metamodel used in DACAR, especially the rule metamodel. Then it will be possible to ensure properties expressed in the model and so to build safe autonomic architectures. We plan to use a metamodeling language such as KERMETA [16] in order to build and reconfigurate real models of both architecture and adaptation policy. We also envisage to integrate an existing ECA-rule execution engine in DACAR, which will interpret properties expressed in the model and apply them while rule execution.

8. REFERENCES

[1] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing Dynamic Reconfiguration in Component-based Systems. *Proceedings of the European Workshop on Software Architectures (EWSA'05)*, pages 1–18, June 2005.

[2] Sara Bouchenak, Fabienne Boyer, Emmanuel Cecchet, Sébastien Jean, Alan Schmitt, and Jean-Bernard Stefani. A Component-based Approach to Distributed System Management - A Use Case with Self-Manageable J2EE Clusters. In *11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[3] Thierry Coupaye and Christine Collet. Denotational Semantics for an Active Rule Execution Model. *2nd International Workshop on Rules in Database Systems, Lecture Notes In Computer Science*, 985:36–50, 1995. London, United Kingdom.

[4] Areski Flissi, Philippe Merle, and Christophe Gransart. A service discovery and automatic deployment component-based software infrastructure for Ubiquitous Computing. *Ubiquitous Mobile Information and Collaboration Systems, CAiSE Workshop, (UMICS 2005)*, June 2005. Porto, Portugal.

[5] Ian Foster and Carl Kesserman. *The Grid: Blueprint for a New Computing Infrastructure.* 2004. ISBN: 1-55860-933-4.

[6] David Garlan, Shang-Wen Cheng, and An-Cheng Huang. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, October 2004. 2004.

[7] Aniruddha Gokhale, Balachandran Natarajan, Douglas C. Schmidt, and al. CoSMIC : An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications. In *Proceedings of the ACM OOPSLA 2002 Workshop on Generative Techniques in the Context of the Model Driven Architecture*, November 2002. Seattle, USA.

[8] Object Management Group. Model Driven Architecture (MDA). Technical Report Document number ormsc/2001-07-01, Object Management Group, July 2001.

[9] Object Management Group. CORBA Components Specification. OMG TC Document formal/02-06-65, Object Management Group, September 2002.

[10] Object Management Group. Deployment and Configuration of Distributed Component-based Applications Specification. Available Specification, Version 4.0 formal/06-04-02, April 2006.

[11] Andreas Hoffman, Tom Ritter, Julia Reznik, and et al. Specification of the Deployment and Configuration. IST COACH deliverable document D2.4, IST COACH, July 2004. http://www.ist-coach.org.

[12] IBM. SCA - Service Component Architecture - Assembly Model Specification. Technical Report Version 0.9, November 2005.

[13] Jeffrey Kephart and David Chess. The Vision of Autonomic Computing. Technical report, IBM Thomas J. Watson, January 2003. IEEE Computer Society.

[14] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26, issue 1:70–93, January 2000.

[15] Sun Microsystems. Java 2 Platform Enterprise Edition Specification. Final Release Version 1.4, november 2003.

[16] Pierre-Alain Muller, Frack Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *Proceedings of MODELS/UML'2005*, pages 264–278, October 2005. Montego Bay, Jamaica.

[17] Stephan Reiff-Marganiec and Kenneth J. Turner. Feature Interaction in Policies. *Computer Networks 45*, pages 569—584, March 2004. Department of Computing Science and Mathematics, University of Stirling, United Kingdom.

[18] Mark Weiser. The Computer for the 21st Century. *Scientific American*, pages 94–100, September 1991.