Linking Reference and Runtime Architectures in Autonomic Systems

Etienne Gandrille[1,2], Catherine Hamon[1] and Philippe Lalanda[2]

[1]France Telecom
Orange Labs Research
28, Chemin du vieux chêne
38 243 Meylan cedex
France

[2]Grenoble University
220, rue de la Chimie
Campus Universitaire
38041 Grenoble, cedex 9
France

Abstract

Autonomic computing seeks to render computing systems as self-managed. In other words, its objective is to enable computer systems to manage themselves so as to minimise the need for human input. Such systems are today of higher interest in many domains, including defence systems. As a matter of fact, the overall complexity of many such software systems is just overwhelming for administrators and some level of automatisation is needed to alleviate their work. It turns out that software architecture can play a prominent role in autonomic computing. Concretely, architectures can be used in order to express constraints to be maintained all along the execution of a system and, conversely, to present the state of the running system.

The currently preferred strategy of researchers and practitioners is to build a number of separate models, including, for instance, a model of the running software architecture, a model of the computing environment, and models focusing on relevant non-functional properties like security, performance, etc. However, the models to be represented, their level of abstraction, their formalisation are still defined on a case-by-case basis. In this paper, we present an approach where explicit reference architectures are maintained during execution and explicitly traced to the design architecture. We show how concepts of design time and runtime can be linked and exploited by an autonomic manager. This approach is validated on a real mediation software architecture used in pervasive environments.

1.0 Introduction

In 2001, Paul Horn, IBM Senior Vice President of Research, introduced the Autonomic Computing Initiative in response to the ever-growing complexity of integrating, managing, and operating computing systems; with the ultimate goal to develop self-managed systems. In a keynote presentation addressed to the National Academy of Engineers at Harvard University in October 2001 and in an accompanying publication [1], Paul Horn alerted the industrial and the academic communities about a looming crisis arising from the ever-increasing complexity of managing computing systems, and qualified it as “our next Grand Challenge”.

Indeed, it clearly appears that software systems are more and more difficult to develop and manage. This is due to a number of reasons, including the following ones:

- Code size. Software systems are often coded by hundreds of developers and end up with millions of lines of code. This is partly due to the incredible progresses in hardware, software systems that we have witnessed in recent years.
- Number of users. Some systems are used by millions of users, over Internet for instance. It is estimated today that some 2 billion people use the Internet, and 4 billion people have a mobile phone subscription. This is an incredible scale to be supported by modern applications.

- Data size. Today, incredible amount of data are produced everyday and made available by new players like Facebook, Twitter, etc. Sensors blended in our living environments also regularly produce tons of data.
- Application nature. New types of applications are emerging. For instance, pervasive applications are becoming reality in homes, buildings, cities, plants, etc. These new applications present stringent new requirements in terms of dynamicity, heterogeneity, etc.

In Brian Fitzgerald's terms [2], we are now facing what he calls "Software Crisis 2.0". This term means that software approaches developed in the past years are now unable to deal with new business demands in terms of development, deployment, and maintenance. We believe that new techniques and new initiatives are needed to push software engineering into runtime in order to better help administrators out. But it is readily admitted that relatively little effort has been dedicated to the deployment and maintenance activities. For a long time, the Software Engineering activity focused on the development phase, seeking to lower problems appearance at runtime.

We believe that autonomic computing is a major initiative in the good direction. This approach, seeking to render computing systems as self-managed, relies on design and runtime models of the system to perform self-management actions. Such models often include representations of the system software architecture. Unfortunately, such representations are often kept separate (i.e. they ate isolated in the design and runtime phases), which makes autonomic reasoning harder to implement and test. In this paper, we present an approach where explicit architecture models are built during execution and constantly and explicitly traced back to some design-level architecture.

This paper is organized as it follows. In the coming section, we provide some background about autonomic computing. Then, in the next section, we focus on autonomic practices based on models and software architectures. In section 4, we develop our proposal based on a traceability links between design and runtime architectures. Then, we sketch an illustration of our approach on Cilia, an autonomic Enterprise Service Bus developed by the Grenoble University in collaboration with Orange Labs, in the context of pervasive computing.

2.0 Autonomic Computing

Autonomic computing [3] takes inspiration from the human body and its Autonomic Nervous System (ANS), which controls many of the muscles and organs. It always ensures that complex biological systems such as vision, digestion, or respiration work in harmony to maintain a steady internal state called homeostasis. In response to variations regarding for instance temperature, posture, food intake, or stress, the ANS adjusts the body functions without often reaching consciousness. Without the ANS, we would be occupied all the time with adapting our vital functions to its needs and to varying environmental conditions. In that spirit, IBM suggested that Autonomic Computing should have autonomic behaviours as observed on the ANS. Complex computing systems should manage themselves with minimal conscious human intervention. It should adjust its internal function to varying circumstances and conditions, hence freeing users from the inherent complexity and the costly management tasks.

To fulfill this vision of autonomic computing, IBM researchers have established an architectural framework, which organizes an autonomic computing system into building blocks that can be composed together to form self-managing systems. Accordingly, an autonomic computing system is organized around a MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop. The MAPE-K loop can be seen as an advanced control loop with two main parts. The first one, called the Managed Element, represents the controlled part of the managed system (e.g., software component, database, or a web server). The second part, called the Autonomic Manager, analyses the situation and takes appropriate actions if needed,

with respect to high-level objectives. This architectural approach is illustrated by figure 1.

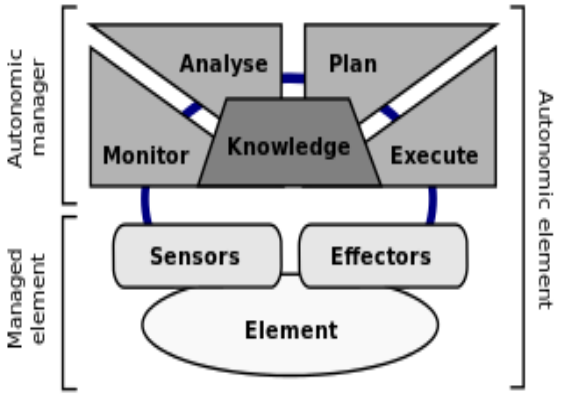

Figure 1. Autonomic computing element.

At the managed element level, Sensors collect information while Effectors carry out and perform changes to the managed element. If we take the example of a web server to be managed autonomically, sensors would collect data about the server load and number of live connections for instance, while effectors would change configuration properties of the server. At the Autonomic Manager level, data collected by sensors are used to construct useful monitoring data. This data is analysed and actions are planned depending on the already acquired knowledge and on the user specified goals and objectives. Actions are executed through effectors. (Notice also that this reference architecture has also been used for other domains like the Organic Computing [4] and its observer/controller architecture). For each stage of the MAPE-K loop, different techniques and tools can be used depending on the specificities of the considered system.

Depending on the targeted system's complexity and degree of autonomicity, different levels of knowledge and reasoning are required to drive autonomic management. As knowledge about the system and its context is shared amongst all the phases of the loop, it is shown as a cross-cutting aspect.

3.0 Architecture-based knowledge

Let us now focus on the notion of knowledge, which is central to advanced autonomic systems. An important question rapidly comes up about knowledge: what should be explicitly represented? (And, conversely, what knowledge should be kept in the problem solver in the form of algorithms?). Generally, the more information is made explicit, the better. This allows the MAPE tasks to focus on the computation. This makes the code leaner, more focused, and easier to change. It is then not surprising that making knowledge explicit is today a strong tendency in autonomic computing. In particular, this knowledge is more and more expressed as architectural models that are understood by the autonomic managers (the smart part of self-managed systems).

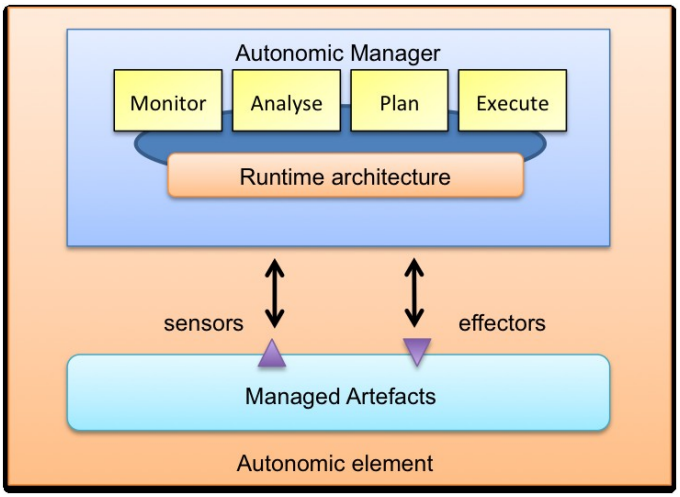

Figure 2. Runtime architecture

An important advantage of the architectural model-based approach is that, under the assumption that the architecture correctly mirrors the managed system, the architectural model can be used to verify that system integrity is preserved when applying an adaptation. This is because changes are planned and applied to the model first, which will show the resulting system state including any violations of constraints or requirements of the system present in the model. If the new state of the system is acceptable, the plan can then be executed on the actual managed system, thus ensuring that the model and implementation are consistent [2].

An interesting approach is to maintain a correspondence between target architecture and the observed architecture (also called the runtime architecture). A runtime architectural model describes the interconnections between components and connectors, and their mappings to implementation modules, as well as constraints on the different possible configurations [5]. The mapping enables changes specified in terms of the architectural model to effect corresponding changes in the implementation.

For instance, the Acme adaptation framework [6,7] uses an architectural model for monitoring and detecting the need for adaptation in a system. The components and connectors of their architectural model can be annotated with a property list and constraints for detecting the need for adaptation. A first-order predicate language is used in Acme to analyse the architectural model and detect violations in the executing system. An imperative language is then used to describe repair strategies. Similarly in C2/xADL [8,9], the difference between an old architectural model and a new one based on recent monitoring data is computed to create a repair plan. Given the architecture model of the system, the repair plan is analysed to ascertain that the change is valid (at least at the architectural description level). The repair plan is then executed on the running system without restarting it.

4.0 Approach

Our approach is in line with the solutions described here before. Indeed, we seek to structure the knowledge of an autonomic manager (the K in the MAPE-K pattern) around the notion of software architecture. Specifically, different architectures are explicitly defined and traceability links are built and maintained at execution time. Architectures come from different steps of the software lifecycle, design and execution. The mapping we propose, enabling rich reasoning capabilities, is however more precise than in most existing works.

We have defined the following types of architectures:

- Design architectures are defined by domain architects and represent the target structure of a system. They are based on the notion of component, connectors, and constraints. They also include some variability, making room for future runtime adaptation (autonomic or not).
- Runtime architectures are built from monitoring data and represent the architecture of a system under execution. They are still made of components and connectors but the notion of variability and constraints is not present anymore. Runtime architectures exhibit runtime phenomenon modelling the current execution.

Design architectures are said to be concrete when they include no variability. They are said to be abstract when some variability is left, for instance regarding the precise component implementations to be used. These architectures are constrained by conformity relationships, as it is made explicit by figure 3. In particular, conformity here means that the architecture at runtime meets all the constraints, structural and behavioural, expressed by the design architecture. Since the design architectures include some variability, conformed runtime architectures can be numerous. They also evolve all the time due to execution contingencies, which is not the case for the design architectures that are stable.

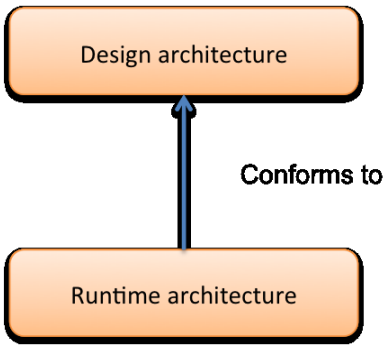

Figure 3. Relationship between architectures.

Variability in design architecture is expressed through the following mechanisms:

- Cardinality constraints. Connections between software components can be optional, unary or n-ary (that is, several instance of a component can be linked to a given component). Thus, the final structure of the system in terms of components and connectors can take different form.
- Abstraction. Some components can be defined through at a high level of specification. They are said to be abstract in that case.

It is the role of a deployer or of an advanced execution machine to determine an appropriate architecture from the design architecture. Decisions to be made concern the cardinality and the abstraction aspects.

These architectures are based on the notions of abstract components, implemented components and runtime components. These entities shared on number of properties, but they are actually very different. They come from different phases of the software lifecycle, they seek to meet different objectives, and, non-surprisingly, they are characterized by different properties and constraints.

The different types of components we are interested in and their relationships are presented in figure 4.

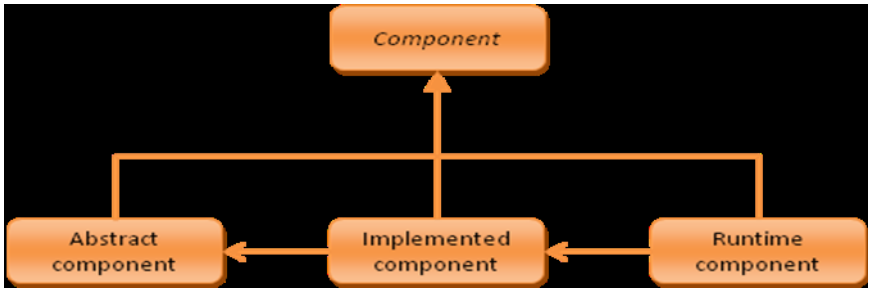

Figure 4 Relationship between components

Let us go through the different types of components.

In the most general way, a component can be seen as a black box with a name, and typed input and output ports. By design, any component is ready to be integrated in architectures, using connectors.

Abstract components are intended to be used in design architectures. Like any component, they define typed input and output ports. They also declare a set of parameters and properties, which have to be used and set in component implementations. Parameters allow setting and changing the component behavior at runtime through configuration. Some of them can be declared in abstract component description, others in implemented components only. Properties are *(key, value)* pairs, describing implementations aspects. Defining properties in an abstract component requires implemented components to define a value for each property. When building the runtime architecture from the design one, it is thus possible to select abstract component implementations that satisfy some property values.

An implemented component is a piece of code. It can be linked to an abstract component, which defines a minimum set of features to be realized (ports, properties, parameters).

A runtime component is an instance of an implemented component running on an execution platform. It has:

- An API for modifying parameter values,
- An API for querying component state: number of processed messages, average process time per message, etc.

These two API implement at the component level the sensors and effectors required by the autonomic MAPE-K loop.

5.0 Conclusion

This work is applied in the Medical project (http://medical.imag.fr), funded by the French government. More precisely, it is applied to the Cilia framework, an autonomic, open source mediation framework [10] developed by the Adele team. Its purpose is to simplify the work of integrators by offering a well-defined and limited set of abstractions to support design, composition, deployment and execution of a variety of mediation chains.

The Cilia framework ((https://github.com/AdeleResearchGroup/Cilia) takes the form of a domain-specific component model, including a specification language and a flexible execution environment. A Cilia component is called a mediator. Its purpose is to realize a single mediation operation like a data

transformation, a security function, an aggregation, etc. A Cilia component is characterized by a number of typed input and output ports. Input ports receive the data to be treated whereas the output ports forward the results of the mediation processing. Ports are the means to connect mediators and, thus, form mediation chains [3].

In the Medical project, we have defined the notions of abstract mediator, implemented mediator, and running mediator (and the associated notions of design and runtime architectures as explained in this paper). Such approach allows the autonomic modification of mediation chains. For instance, based on architectural information, we can successfully change mediator's implementation or add new mediation branches.

6.0 References

[1] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology", IBM, 2001

[2] Fitzgerald, B., "Software Crisis 2.0", Computer, April 2012, Volume: 45 , Issue: 4 pp. 89 - 91

[3] P. Lalanda, J. MaCann and A. Diaconescu, "Autonomic Computing : Principles, Design and Implementation", Springer Verlag, to appear in May 2013.

[4] Urban Richter, Moez Mnif, Jürgen Branke, Christian Müller-Schloer, Hartmut Schmeck,"Towards a generic observer/controller architecture for Organic Computing.", pp. 112-119, 2006.

[5] P. Oreizy, N. Medvidovic, R. N. Taylor: "Architecture-based runtime software evolution", Software Engineering, 1998. Proceedings of the 1998 (20th) International Conference on, pp. 177-186, 1998.

[6] Garlan, D. and Schmerl, Exploiting architectural design knowledge to support self-repairing systems. In Proceedings of the 14th international conference on Software engineering and knowledge engineering, 2000.

[7] Garlan, D. and Schmerl, B. Model-based adaptation for self-healing systems. In Proceedings of the first workshop on Self-healing systems, 2000.

[8] Oreizy, P., Medvidovic, N., and Taylor, R. N. Architecture-based runtime software evolution. In ICSE '98: Proceedings of the 20th international conference on Software engineer- ing. IEEE Computer Society, Washington, DC, USA, 177–186, 1998.

[9] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. Towards architecture-based self-healing systems. In Proceedings of the first workshop on Self-healing systems, 2002.

[10] I. Garcia, G. Pedraza, B. Debbabi, P. Lalanda and C. Hamon. Towards a service mediation framework for dynamic applications. In Proceedings of the IEEE 2010 Asia-Pacific Services Computing Conference, 2010-12-06, Hangzhou, China.