Maintaining Traceability Links between Design and Runtime Architectures to support Autonomic Management

Philippe Lalanda*, Stéphanie Chollet†, Etienne Gandrille*‡, and Catherine Hamon‡
*Université Grenoble Alpes, LIG
F-38041, Grenoble
firstname.name@imag.fr
†Université Grenoble Alpes, LCIS
F-26900, Valence
stephanie.chollet@grenoble-inp.fr
‡Orange Labs
F-38243, Meylan
firstname.name@orange.com

***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. Software 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. In this paper, we present an approach where design and runtime architectures are used to manage service-oriented systems. We show how concepts of design time and runtime can be linked and exploited by an autonomic manager or by a human administrator. This approach is validated on a real use case belonging to the pervasive domain.**

I. Introduction

The software engineering community has been striving for years to deliver software as quickly as possible while preserving the possibility to adapt it gracefully afterwards. This is still true today. The challenge is even exacerbated by always fiercer competition, rapidly changing market conditions, drastically reduced time-to-market, and the emergence of very demanding domains. Pervasive computing [1], for instance, requires the development of highly flexible applications that have to remain functional in very dynamic environments.

To satisfy such requirements, Service-Oriented Computing (SOC) has appeared recently [2]. The very purpose of this reuse-based approach is to allow application development through late composition of independent software elements, called services. Services are described and published by service providers; they are chosen and invoked by service consumers. This is achieved within Service-Oriented Architectures (SOA), providing the supporting mechanisms for publication, discovery, binding, etc.

Obviously, service orientation brings interesting features that have made it popular in many domains. If carefully planned, using services can effectively support the development of pervasive applications, including health applications. First, weak coupling between consumers and providers reduces dependencies among composition units, letting each element evolve separately. Late-binding and substitutability then improve adaptability: a service chosen or replaced at runtime is likely to better fulfil expectations.

However, as said earlier, the service-oriented approach brings up maintenance problem. It can be difficult for administrators to follow the architectural evolution of an application and, above all, to check its validity. As a matter of fact, the structure of an application depends on the context (the available services and the corresponding quality of service) and cannot be approved in advance (at deployment time for instance). Also, decisions regarding services selections and bindings are taken by the system itself and the administrator has little control over it. It is then of utter importance to provide appropriate information to the administrators so that he/she can intervene on the system when needed. This information has to be of high level in order to be understandable and usable by the administrator.

Autonomic computing seeks to render computing systems as self-managed [3], [4]. In other words, its objective is to enable computer systems to manage themselves so as to minimise the need for human input. Autonomic computing relies on the construction of 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 not traced back to design decisions, which often makes autonomic reasoning hard to implement and test. The reason is that much knowledge has to be inserted in the autonomic problem-solving algorithms and is not made explicit. Such approach clearly does not favor code understanding and system evolution.

In this paper, we present an approach where explicit architecture models are built both at design time and 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 present our motivating scenario in the pervasive health domain. In the third section, we present our approach and the corresponding formalization. The motivating example is then developed with our approach for

2014 IEEE Emerging Technology and Factory Automation (ETFA)
978-1-4799-4845-1/14/$31.00

validation purposes. This paper ends with the presentation of some ongoing and coming work.

II. Scenario

Let us first introduce our motivating scenario, which has been defined within a collaborative project called Medical[1]. This use case corresponds to situations commonly encountered in the health industry when it comes to home care. Precisely, the service we are working on is called actimetrics and is based on measurement and analysis of motor activities of a subject in his environment. Its purpose is to track and memorize movement patterns of inhabitants in order to rapidly detect abnormal changes. Indeed, for elderly or patients, behavioural changes at home can be a sign of more serious underlying problems.

This service relies on localisation information collected in homes from a network of heterogeneous and dynamic sensors. Any kind of sensors can be used to do so including presence detectors, pressure sensors, body localizers but also events emitted by any electronic device like a TV, an oven, a washing machine, a coffee machine, etc. Collected data is regularly transferred in an appropriate format to a remote IT server that builds analysis matrices. Such matrices are then used to detect behavioural deviances, which are presented to the doctors.

In technological terms, this use case heavily relies on the notion of software service, which is rather common today. Devices and applications are exposed as software services of different natures, including UPnP[2], Web Services[3], and DPWS (Device Profile for Web Services). Implementing the use case requires integrating such services exposed on local or wide-area networks.

Implementing the actimetrics use case in an open world, that is not limited to a fixed set of statically selected devices, turns out to be very challenging. This is due to a number of reasons. First, the code needed to integrate different sources of information is complex and highly error prone. In particular, the level of synchronisation to be achieved brings significant difficulty. Also, it is difficult to effectively deal with the dynamicity and heterogeneity of devices. All situations cannot be anticipated at design time and some intelligence is needed at runtime to cope with changing environments.

Let us have a closer look at these requirements. Dealing with dynamicity is necessary because the lifecycle of services is beyond the applications control. In other words, their availability is not managed by the applications. Platform-level mechanisms are then needed in order to be constantly aware of services availability and to allow runtime modifications. Application-level mechanisms are also required so that applications can react gracefully to availability evolutions. For instance, strategies are needed to specify when and how a service can be changed or replaced at runtime. Heterogeneity is another critical challenge. Indeed, a number of technologies have been proposed in the last few years to implement software service. These technologies are hard to integrate: they use different description languages, different notification patterns, and different invocation styles. Web Services, for instance, use WSDL (Web Service Description Language) for description and the SOAP protocol for invocation. Other technologies, by contrast, respectively use Java-like interfaces enriched with metadata and mere Java method calls.

[1]See http://medical.imag.fr.
[2]See http://www.upnp.org.
[3]See http://www.w3c.org.

III. Approach

A. Overview

In order to ease administrators tasks, we developed an architecture-based autonomic approach. Specifically, we propose to formalize and constantly maintain the following types of architectures:

- **Design Architecture.** Such architecture is defined by domain architects and specifies the target structures of a (service-oriented) system. Design architectures generally include some variability, making room for future runtime adaptations. For instance some variability may be left regarding the component implementations to be used. Such architectures are said to be abstract.
- **Deployment Architecture.** Such architecture is configured and ready for deployment on a target client site. Deployment architectures may also retain some variability for runtime adaptation, which makes sense in the service-oriented approach where some decisions have to be taken at runtime. They are however more constrained than design architectures.
- **Runtime Architecture.** Such architecture is built from monitoring data and formalizes the architecture of a system under execution (at a given level of abstraction). Here, the notions of variability and constraints are not used anymore. Runtime architectures are evolving frequently, as naturally expected with service-based applications.

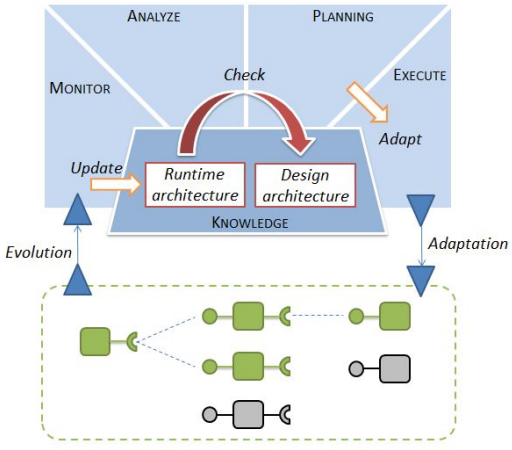

Fig. 1. Overall approach.

Figure 1 illustrates our approach. It shows that the design and runtime architectures and their relationships are part of the knowledge used by an autonomic manager. Here, the autonomic manager is classically structured around a MAPE-K loop as introduced by IBM [5], where M is for Monitoring,

A for Analysis, P for Planning, E for Execute and K for Knowledge.

B. Formalisation

These three types of architecture are major artifacts, defined and used all along software lifecycle. It is worth noting that they rely on the same core concepts (components, bindings, etc.) but with slightly different meanings. This is why we decided to define a structured meta-model made of four parts:

- an *architecture meta-model* defines the concepts shared by all the architecture types, at a high level of abstraction.
- a *design architecture meta-model* refines and extends these concepts for the design steps.
- a *deployment architecture meta-model* refines and extends these concepts for the deployment step.
- a *runtime architecture meta-model* refines and extends these concepts for the runtime steps.

This modeling organization is illustrated by Figure 2.

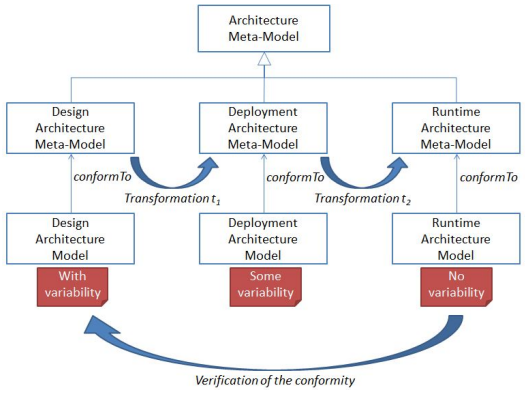

Fig. 2. Meta-models organization.

It is to be noted that architectures are derived from one another, as made explicit by Figure 2, through transformations. That means, among other things, that deployment and runtime architectures meet all the constraints, structural and behavioral, expressed by the design architecture. Since the design architecture may include some variability, correct deployment and runtime architectures can be numerous. The runtime architectures also evolve all the time due to execution contingencies, which is not the case for the design architecture, which remains the same.

Figure 3 describes the global architecture meta-model. It contains the core concepts used to depict software architecture. An architecture is composed by components and bindings. The components can be enriched with definitions. There are three kinds of valuable definitions: property definition, parameter definition, state variable definition. The main concepts are abstract. In each sub-meta-model, they are overloaded.

One of the major interest to keep the main concepts in a global meta-model is that these particular concepts can be use and refined in the different sub-meta-model, which are actually simply specialized for each step of the lifecycle. Thus, this solution guarantees the continuity for these concepts, which is not the case with an unique meta-model. Il also requires fewer rules definitions, generally complex (such as in OCL).

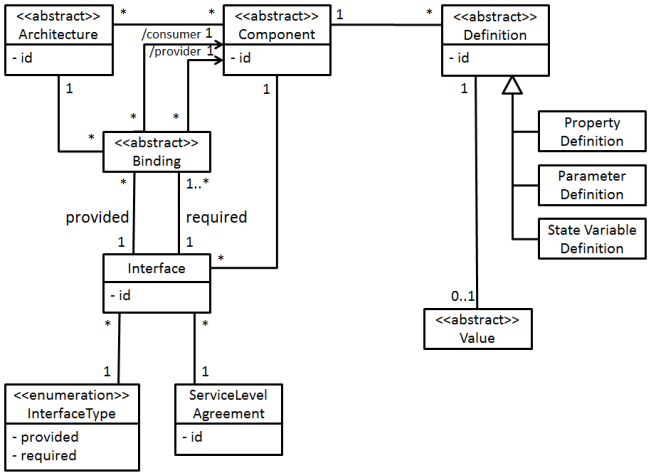

Fig. 3. Architecture meta-model.

The design meta-model defines the vocabulary and the grammar used at design time. As explained in Figure 1, the design meta-model inherits from concepts of the global meta-model. We have introduced the element required to express design architectures. Figure 4 presents the design meta-model. A design architecture can be constrained. The components of the architecture are either specifications or implementations. An implementation may correspond to a specification. Bindings between components are expressed with cardinalities (minimum and maximum for provided and required components).

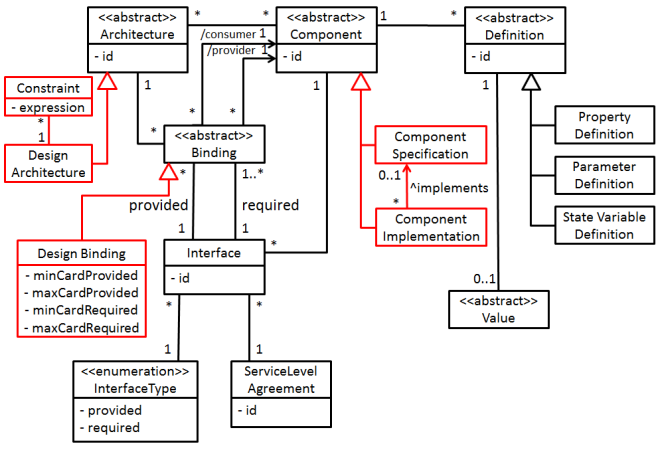

Fig. 4. Design architecture meta-model.

This architecture contains variability constraints mainly expressed by the cardinalities for the bindings. Figure 5 illustrates a design architecture. It contains two components (specification and implementation) with a binding expressed with cardinality constraints. In the service level agreement, there is information for the binding between the components.

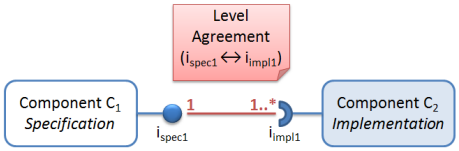

Fig. 5. Example of design architecture.

Variability is a key point in design architectures. Design architectures have to specify common structures and, also, leave space to bring in specific characteristics at runtime. Emerging tools have introduced the possibility to express variations in architecture in a systematic way. This has to be considered in two dimensions, space and time. The locations in design artifacts, where a specific decision has been narrowed to several options, are called variation points [6]. Variability modeling comes down to the identification and documentation of those variation points to facilitate specific configuration activity at runtime. However, the identification and explicit representation of variation points is a thorny problem. It requires, not only to introduce variation points and their relative variants, but also to define some dependencies among variation points and to express possible context-related restrictions.

The deployment architecture is the final refinement, if any, of the design architecture. This deployment architecture contains the structures, behaviors and constraints to be met by the application at hand, given the target platform and the quality-related goals. It is presented in Figure 6.

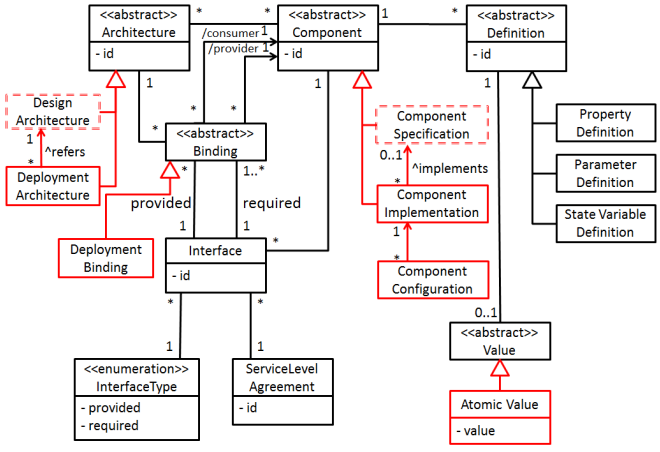

Fig. 6. Deployment architecture meta-model.

The runtime architecture, which meta-model is presented by Figure 7, stores runtime information about the running software and the platform under operation. This architecture provides a model of runtime phenomena, with trends and past data, and is intended for use by autonomic managers or by administrators. Using this knowledge module is a very convenient way for domain engineers to create autonomic managers. Managers use high level APIs provided by the knowledge module to get relevant information and trigger adaptations.

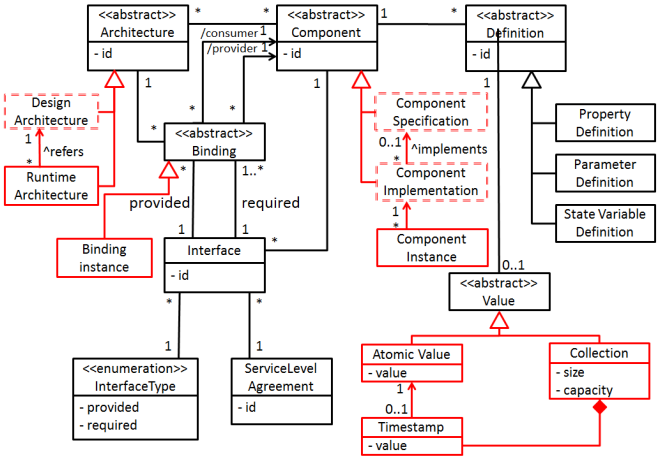

Fig. 7. Runtime architecture meta-model.

Traceability links between design and runtime architectures are constantly maintained. To do so, we used and extended graph-based algorithms. Indeed, architectures can be formalized as graphs, where components (specification, implementation or under execution) are nodes and links between components (specification or real connections) are arcs. Presentations of the algorithms are beyond the scope of this paper. It is worth mentioning, however, that an important issue is the synchronisation issue. Traceability algorithms have to be started when all the ongoing modifications are done, as in transactions, otherwise results have no sense. When a runtime architecture structure cannot be traced back to a design decision, a warning is generated for the autonomic manager.

IV. Validation

Let us now return to the actimetrics use case introduced in Section II. It has been implementing following the architecture presented by Figure 8. It is based on an iPOJO runtime on a gateway and JMS communications with a Web Service placed on a remote server. This is where behavioral analysis and pattern detections are done. The purpose of the mediation code is to collect, aggregate and transform as much information as possible regarding inhabitants movements. This information can come from all devices placed in the house, albeit heterogeneous ones.

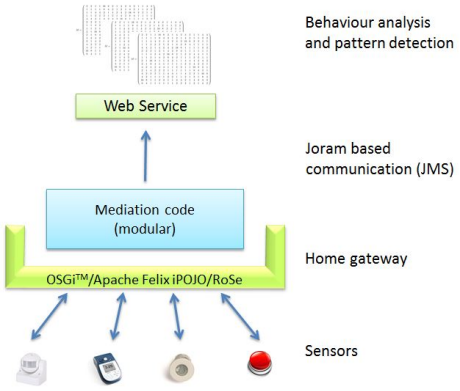

Fig. 8. Actimetrics application architecture.

As illustrated by Figure 9, the design mediation chain of the actimetrics use case is made of the following elements:

- A branch made of the mediators *push-button*, *button-enrich*, *button-translate*. This branch gets information from a special purpose button that can be pressed by the inhabitant, then enriches the captured information and translates it into a standard format.
- A branch made of the mediators *motion-sensor*, *motion-enrich*, *motion-translate*. This branch gets information from any kind of sensors able to detect a movement, then enriches the captured information and translates it.
- A branch made of the mediators *set-top-box*, *stb-enrich*, *stb-translate*. This branch gets information from set-top-boxes when used by the inhabitant, then enriches the captured information and translates it.
- A branch made of the mediators *filter*, *enrich*, *transform*, *translate* and *event-webservice*. The filter mediator receives data from different devices (from the corresponding branches), synchronizes them and eliminates redundancy or dubious information. Then, data are enriched with information like the house id, transformed, translated and sent to a Web Service.

The first three branches (button, motion, set-top-box) are optional. They depend on the presence of devices that are not known at design time. The fourth branch is always instantiated, regardless of the environment and runtime conditions.

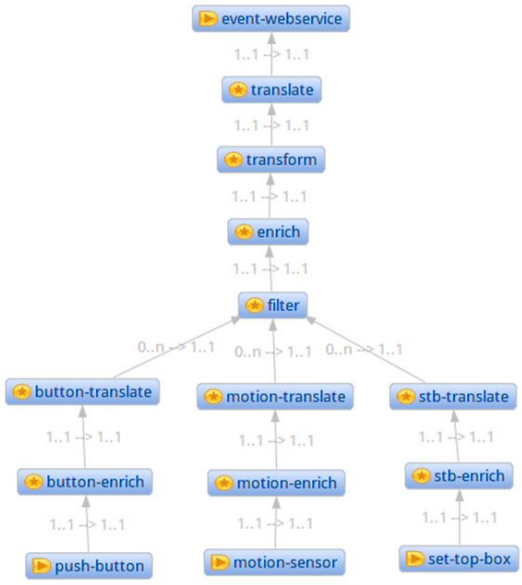

Fig. 9. Design mediation chain (tool capture).

An instance of runtime architecture is presented by Figure 10. In this figure, state variables are not presented (they are in fact accessible in specific windows of our tool). It appears that, among the optional branches, two of them have been instantiated. These are similar branches that have been dynamically created to deal with buttons discovered in the environment. Here, branches are duplicated to separately integrate the two detected buttons. An alternative solution would have been to create a single mediation branch dealing with all the possible buttons detected in a house. Our tool traces runtime decisions back to the design space, even if these decisions are made autonomically, either by the runtime framework or by an autonomic manager.

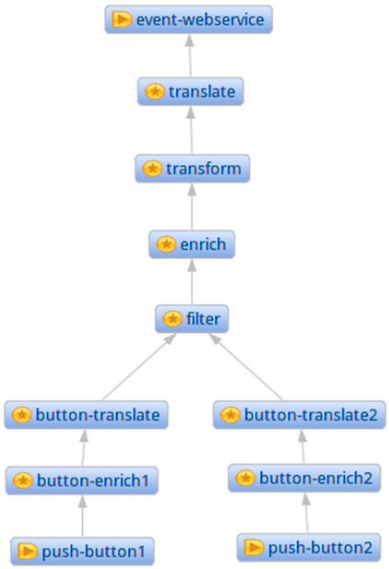

Fig. 10. Runtime mediation chain (tool capture).

As shown by Figure 11, the two branches dealing with concrete buttons in the runtime architecture are linked to the design mediation chain. Precisely, they are linked to the optional branch addressing press buttons.

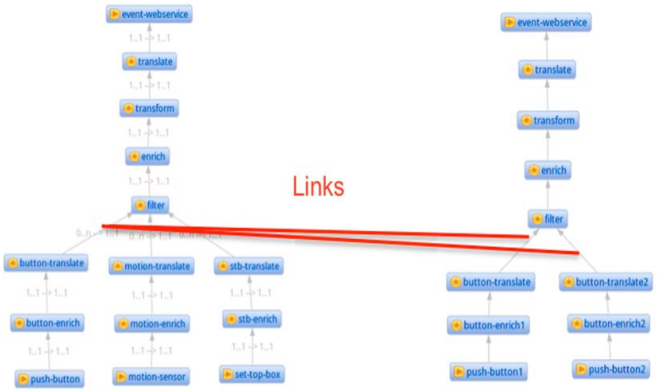

Fig. 11. Traceability links (tool capture).

To illustrate this, let us get back to our actimetrics example. If, at runtime, a button is deficient and causes problems (for instance, the base branch waits too long for its inputs), then the administrator knows that he/she can suppress the branch without affecting the global mediation application. This is indeed expressed by the design mediation chain where the ”button branches” are specified as optional. In spite of the system dynamicity due to services arrival and departure, the

administrator is able to take a good decision in line with the designers plans.

V. Related work

In autonomic computing and adaptive systems, requirements models [7], [8] and architecture models [9], [10] have been used for self-adaptation driven by traceability links. The notion of knowledge is actually central to autonomic computing. An important challenge is to figure out what should be explicitly represented. Of course, the more information is made explicit, the easier it is to implement and maintain the autonomic control loops. The code is made leaner, more focused, easier to change and even reusable in certain cases. It is then not surprising that making knowledge explicit is today a strong tendency in autonomic computing.

An important advantage of this architectural approach is that 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 [11].

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. The mapping enables changes specified in terms of the architectural model to effect corresponding changes in the implementation. For instance, the Acme adaptation framework [12], [13] uses an architectural model for monitoring and detecting the need for adaptation in a system. Similarly, in C2/xADL [11], [14], the difference between an old architectural model and a new one based on monitoring data is computed to create a repair plan.

A limit of the existing approaches is related to the expressiveness of the target architectural model. Here, architectures are specified precisely through component, connectors and constraints to be maintained and there is relatively few places for adaptation. That is, these architectures do not include much variability. Another shortcoming concerns the lack of tools managing the whole software life cycle. That is, there is a crucial for architectural tools helping developers and administrators to model architecture at design time and at runtime. Most of the time, these artifacts are developed or modeled with different tools and kept separated.

VI. Conclusion

Service-oriented computing allows postponing architectural decisions to runtime, that is to say when the execution conditions are entirely known. Such a capability is extremely important in a number of domains, like pervasive computing for instance, where execution environments are very changing and unpredictable. The downside is that such systems are hard to manage. Indeed, the software architecture of those systems is subject to unpredictable changes that are difficult to follow and understand for administrators. It is all the more challenging for them to bring changes while preserving the system coherence and integrity.

In this paper, we propose to present to the administrators a continuous, easily understandable, picture of the design and runtime architectures and of their links. Thus, when a decision has to be made, advanced architectural information is available to guide them. To do so, we had to precisely define the notions of design architecture, variability and runtime architectures. These notions are based on abstract components, concrete components, executed components, service dependencies and bindings with cardinality.

References

[1] M. Weiser, "The computer for the 21st century," in *Human-computer interaction*. Morgan Kaufmann Publishers Inc., 1995, pp. 933–940.

[2] M. P. Papazoglou, "Service-Oriented Computing: Concepts, Characteristics and Directions," in *Proceedings of the fourth International Conference on Web Information Systems Engineering*, Los Alamitos, CA, USA, 2003, pp. 3–12.

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

[4] P. Lalanda, J. A. McCann, and A. Diaconescu, *Autonomic Computing - Principles, Design and Implementation*. Springer–Verlag, June 2013.

[5] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[6] J. van Gurp, J. Bosch, and M. Svahnberg, "On the Notion of Variability in Software Product Lines," in *2001 Working IEEE / IFIP Conference on Software Architecture*. IEEE Computer Society, 2001, pp. 45–54.

[7] A. M. Elkhodary, N. Esfahani, and S. Malek, "FUSION: a framework for engineering self-tuning self-adaptive software systems," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2010, pp. 7–16.

[8] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy Goals for Requirements-Driven Adaptation," in *RE 2010, 18th IEEE International Requirements Engineering Conference*. IEEE Computer Society, 2010, pp. 125–134.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[10] J. Floch, S. O. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven, "Using Architecture Models for Runtime Adaptability," *IEEE Software*, vol. 23, no. 2, pp. 62–70, 2006.

[11] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-Based Runtime Software Evolution," in *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering*. IEEE Computer Society, 1998, pp. 177–186.

[12] B. R. Schmerl and D. Garlan, "Exploiting architectural design knowledge to support self-repairing systems," in *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, 2002, pp. 241–248.

[13] D. Garlan and B. R. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the First Workshop on Self-Healing Systems*. ACM, 2002, pp. 27–32.

[14] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "Towards architecture-based self-healing systems," in *Proceedings of the First Workshop on Self-Healing Systems*. ACM, 2002, pp. 21–26.