Design and runtime architectures to support autonomic management

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

[1] Orange Labs
28, Chemin du vieux chêne
38243 Meylan cedex, France
firstname.name@orange.com

[2] Grenoble University
220, rue de la chimie
38041 Grenoble, cedex 9, France
firstname.name@imag.fr

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 [5,6]. 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 health domain and built with the Orange Labs.**

Keywords—autonomic; services; design, runtime architecture.

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 unpredictable and dynamic environments.

To satisfy such requirements, development paradigms are regularly proposed. Component-Based Software Engineering [2], for example, allowed practitioners to build software of better quality in lesser time. Service-Oriented Computing (SOC) has appeared more recently [3,4]. 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.

Actually, a service has multiple facets. From the consumer point of view, a service is perceived as a specification made publicly available. Such specification, hiding implementation details like the programming language or the supporting platform, is the basis for runtime services selection. Once the selection has been made, a service is viewed as an instance that is programmatically called by the consumer, or as an implementation that has to be instantiated and managed. Some approaches allow substitutability: a service instance can be transparently replaced at runtime by another one, as long as they both implement the same specification.

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 the consumer 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 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 [5,6]. 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. As a matter of fact, the overall complexity of many software systems is just overwhelming for administrators and some level of automatisation is needed to alleviate their work. We believe that this is actually the case today for the maintenance of service-oriented systems, especially in new demanding fields like pervasive computing for instance.

An autonomic system is structured as a set of autonomic elements that may collaborate or not. Each element implements a control loop monitoring and adapting a set of managed artifacts (or resources) and is based on the notion of autonomic manager. A managed resource represents any software or hardware resource that is endowed with autonomic behavior by coupling it with the autonomic manager. Managed resources provide specific interfaces, called *touch points*, for monitoring and adaptation. Two types of control points have to be differentiated: *sensors* and *effectors*. Sensors, often called probes, provide information about the managed resources. Effectors, provide facilities to adjust the managed resources and, then, change their behavior. IBM researchers also established an architectural framework to implement the control loop. This framework is known under the name MAPE-K for Monitor, Analyze, Plan, Execute, and Knowledge. It is a logical architecture that defines the different activities to be carried out.

The notion of knowledge is clearly central in this framework [6]. 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 MAPE tasks. 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. In particular, this knowledge is more and more expressed as architectural models.

Autonomic computing often relies on the construction runtime models of the system. Such models often include representations of the system software architecture. Unfortunately, they 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 runtime architectural models are built during execution and explicitly traced back to some design-level architecture. Design architectures embed variability, which leaves some room for runtime adaptation.

This paper is organized as it follows. In the coming section, we present our motivating scenario in the pervasive health domain. We also present the technology used so far for implementation, that is the Cilia mediation framework. In the next section, the third, we provide some background about autonomic computing and architecture-based solutions. In the fourth section, we develop our proposal based on a traceability links between design and runtime architectures. Then, in section 5, we show how this approach has been applied to Cilia. This realization is done by the Grenoble University in collaboration with Orange Labs, in the context of pervasive computing. The purpose of the last section is to synthetize the presented work and expose future works.

II. MOTIVATING SCENARIO AND USED TECHNOLOGY

A. *Actimetrics use case*

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 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 treating 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 (www.upnp.org), Web services (www.w3c.org), and DPWS (Web services on devices). 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.

Dealing with dynamicity is necessary because the lifecycle of most 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

---

1 The Medical project is funded by the OSEO and the French Ministry of industry. It brings together industrial and academic actors including Orange Labs, Grenoble University, ParisTech and Scalagent.

years to implement software service. These technologies are hard to integrate: they use different description languages, different notification patterns, and different invocation styles.

B. *Implementation with Cilia*

To implement this use case, we use a service-oriented mediation framework. This framework, called Cilia, is available in open source[2]. Its purpose is to simplify the work of developers by offering a well-defined and limited set of abstractions to support design, deployment and execution of data-oriented applications.

Cilia is a domain-specific service-oriented 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 mediator 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. A mediator can also be characterized by a service dependency, resolved at runtime. Here, the dependency is expressed as a Java interface.

The content of the mediators is divided into three java classes: a scheduler, a processor and a dispatcher (see figure 1). The purpose of the scheduler is to store the data received in the input ports and to apply a triggering condition. It deals with all the synchronization issues. When the condition held by the scheduler becomes true, all the data retained by the scheduler are sent to the processor. The processor applies the mediation operation to the transmitted data. The result of this operation is sent to the dispatcher, which places the results in the output ports.

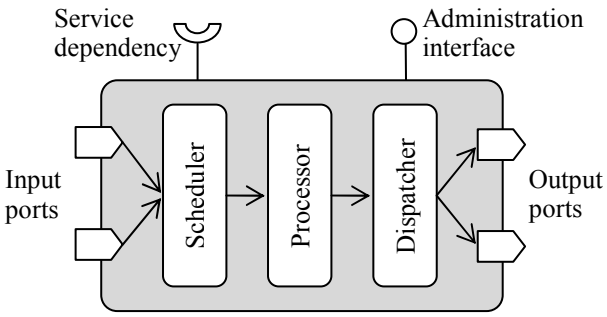

Figure 1. Cilia mediator architecture

A composition of Cilia mediators, that is an architecture, is called a mediation chain. A mediation chain is formed by a set of connected adapters and mediators. Adapters feed the mediators (and the destination resources) with data in the appropriate format and with the appropriate timing. Mediators constitute the heart of the chain since they implement the effective mediation operations.

Mediators (and adapters) are connected via bindings. A binding describes a connection between an output port and an input port. At execution time, a binding is realized by a communication protocol transferring data from a mediator (or adapter) to another mediator (or adapter). This protocol can be specified at deployment time, but also at development time. Cilia supports local and distant communication protocols.

The Cilia execution framework is built on top of OSGi and iPOJO [17], the Apache service-oriented component model. It also includes RoSe, an open source communication middleware that is able to dynamically import and export services[3].

A mediation chain is created in the following manner. A specification file is transmitted to the Cilia runtime. These specifications are transformed into a number of iPOJO components definitions. At least five iPOJO components are created for each mediator: one component for the scheduler, one component for the processor, one component for the dispatcher and two components for the in and out communication ports (more components are created if different protocols are used by different ports). The defined iPOJO components are then instantiated and executed. From this point, the mediation chain is operational (and the desired integration is achieved).

Let us remind here that iPOJO relies on byte code manipulation to create extensible containers encapsulating the execution of Java classes. A container can host a number of handlers implementing non-functional aspects (handlers are triggered before or after a method call). This feature is particularly convenient for implementing dynamic monitoring functions attached to a component.

To sum up, Cilia is a recent service-oriented framework meeting the stringent requirements data mediation. It is and well adapted to the implementation of commonly accepted integration patterns, which favors its acceptation by domain developers. The use of domain-specific concepts also simplifies the creation and understanding of mediation chains. Cilia is currently tested with the Orange Labs and Schneider Electric in order to implement data integration in pervasive applications.

The use of service orientation is particularly effective to integrate dynamically available resources. It raises however important management issues. Indeed, the system changes its structure by itself. As a result, it is pretty difficult for administrators to follow and to check the availability of a given configuration.

Also, adapting Cilia chains to new runtime conditions or users demands still depends on skilled administrators and generally requires some downtime. In many domains, administrators are not available or not skilled enough. Also, service interruption is not always allowed. Solutions must be developed in such cases in order to automate administration or to provide effective help to administrators.

More information about Cilia can also be found in chapter 9 of [6].

[2] The Cilia framework is available at: https://github.com/AdeleResearchGroup/Cilia

[3] The RoSe framework is available at: https://github.com/AdeleResearchGroup/ROSE

III. OUR APPROACH

A. Architecture-based

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

- Design architectures. These architectures are defined by domain architects and define the target structures of a (service-oriented) system. They may include some variability, making room for future runtime adaptation. 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 component implementations to be used.
- Deployment architecture. These architectures are appropriately configured and ready for deployment on a target client site. They may also include some variability for runtime adaptation. They are however more constrained than design architectures.
- Runtime architectures. These architectures are built from monitoring data and formalize the architecture of a system under execution (at a given level of abstraction). They are still made of components and connectors but the notions of variability and constraints are not present anymore.

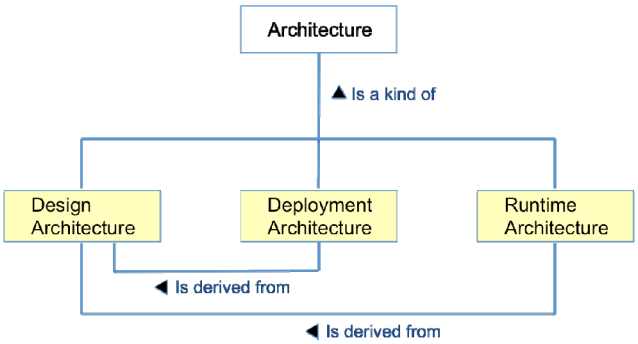

Figure 2. Software architectures.

As illustrated by figure 2, all these architecture are based on the very same notions of components, connectors, and constraints. However, they are defined and valued at different steps of the software lifecycle, from the design phase to the execution time. For that reason, they differ in terms of provided information regarding software components, connectors, and constraints.

These architectures are derived from one another, as made explicit by figure 2. That means that the deployment and runtime architectures meet all the constraints, structural and behavioural, expressed by the design architecture. Since the design architectures may include some variability, correct 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.

B. Variability

Regarding design and deployment architectures, the key point is to express variability. Identifying and describing variability is a hard problem. Many different approaches have actually been proposed to describe, and to resolve, variability. They can be based on natural language, semi-formal languages like UML or feature diagrams, or formal languages like ADLs. Many approaches directly focus on the notion of software architecture.

Design architectures have then 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 [11]. The locations in design artifacts, where a specific decision has been narrowed to several options, are called variation points [12].

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.

A number of approaches to support variability have been proposed. Two of them have retained our attention. First, feature modeling [13] has been used to represent common and variable features among products of a same family. Some extensions permit the representation of additional information regarding specific products, including constraints and dependencies, e.g. a selected feature has a required or excluded relationship with another feature, by means of feature diagrams or cardinality-based feature modeling [14]. Such approaches, supported by adapted development tools, have been pushed by industrial and research organizations.

Pure-systems [15] is a representative platform based on feature-oriented model technology supporting the development of software products families. In this approach, a feature model, made of several feature diagrams, specifies all reusable artifacts and their relationships and dependencies. Features can be mandatory, optional, alternative or following the 'or' semantics. Then, variation description model makes decisions depending on feature types on top of feature model definition and gives more restrictions for describing specific product diagram.

According to the second approach, a group of designer, including domain experts and architects, start out with an investigation of commonalities in a given domain and structure them in software product line architecture. The variability may be modeled structurally at some variation point locations or in term of a generic variability model beside the product line architecture in order to derive a set of dissimilar configured product architecture matching the requirements of a software products family.

C. Architectures description

The design architecture is a key element. It represents the common architectural characteristics shared by all runtime applications. It is made of abstract and concrete components, services, connectors between these components and variation points.

The design architecture can be further detailed into more refined architectures, in order to address more specific functional goals. As such, design architectures can be refined to define additional architectural aspects, such as particular constraints, service types and dependencies. Refined architectures provide better support for developing groups of applications with particular functional or quality-related goals.

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.

Design and deployment architectures are then made of abstract and concrete components. An abstract component or service is a description that is independent of any implementation. It retains the major features of the service orientation and ignores low-level technological aspects. In our architectural model, an abstract service is defined in the following terms:

- Functional interfaces specify the functionalities the service provides. A functional interface can define either a set of operations or a set of ports for data-flow transfer.
- Properties, identified by their names and types, can be divided into three categories. Service properties define static service attributes that cannot be modified when specifying an abstract service composition (e.g. message format property – text or multimedia). Configurable properties represent dynamic attributes used to configure abstract services during the customized service composition process (e.g. destination of messages sent). Quality properties define static or dynamic attributes regarding non-functional aspects of service instances, such as security or logging properties.

A concrete component or service is an implementation of an abstract component or service. For instance, a concrete service can be a Web Service, an UPnP service or a DPWS service. In addition, a concrete service has to implement the abstract service communication pattern (for instance, a publish/subscribe or client/server communication mode). Several concrete services can be made available for a single abstract service.

Runtime architectures exhibit runtime phenomenon modeling the current execution. They are made of components, services and a set of properties characterizing their execution state. Architectural reflexion is needed to ensure that available runtime architectures constantly reflect the managed system state.

In our approach, runtime architectures rely on the notion of state variables that are used to model the dynamics of the running components and services. This approach draws its inspiration from control theory. The traditional goal of control systems is to maintain parameters in a certain threshold. These parameters are usually physical measurements or quantities such as speed and temperature. Control techniques have been developed to meet these needs, which include the use of feedback loops. To achieve this, it is necessary to have an accurate representation of the system [6].

In our case, state variables are attached to global applications but also to their constituents (components, services and bindings). Their values, called measures, are used to assess the quality of service of the components during their execution. The purpose of the autonomic managers is to check that the state variables are within the limits defined by the design architectures, through constraints definitions. Technically speaking, state variables are kept in circular lists in order to keep records of the past. The size of the lists is configurable and can be changed at runtime. Implementation aspects are detailed later on in this paper.

D. Autonomic management

The design architecture guides autonomic managers, which purpose is to maintain correct software architectures. That is, an autonomic manager as to select and instantiate components, bind them together and set configuration properties. It enables to delay design decisions through the use of variation points. In this context, the autonomic manager must ensure that the runtime architecture conforms to the reference architecture. This is illustrated by figure 3.

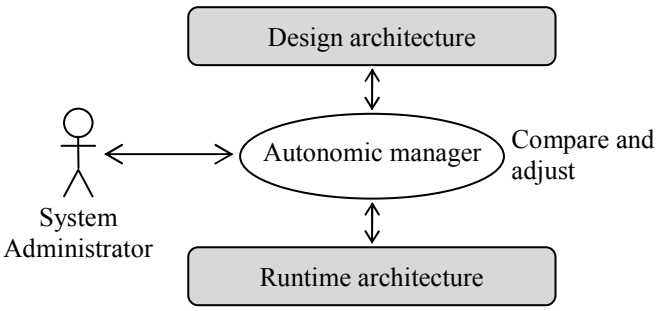

Figure 3. Architecture-based autonomic management.

In autonomic computing, architecture-based approaches were mainly used during the deployment phase. The purpose of such an autonomic *deployer* is to use an architectural model in order to deploy, instantiate and configure components. In most current solutions, however, automatic deployment is executed off-line and later architectural changes require full re-deployment [6]. In order to deal with runtime changes, we progressively pushed the initial off-line reasoning and knowledge into the runtime. This requires more sophisticated reasoning capabilities and knowledge.

This is why we introduced architectural models with variability and state variables, to enable a formal representation of system knowledge. It was also key to create and maintain at runtime relationships between the architectural models.

IV. APPLICATION TO CILIA

We applied the proposed approach to Cilia and to the actimetrics use case (see section 2). To do so, we had to realize the following developments:

- We extended Cilia with touch points, as defined in autonomic computing, in order to dynamically monitor and adapt the mediation chains under execution and some aspects of the supporting execution platform (essentially the service discovery functions).
- We created an architecture-based tool dedicated to designers and administrators. This Integrated Development Environment (IDE) allows the modeling of the design and deployment architectures, and the representation of the runtime architectures. Links between these different architectures are created and maintained. These artifacts can be used by an autonomic manager or by an administrator to perform maintenance operations.

A. Architectural touchpoints

As illustrated by figure 4, the Cilia framework now provides a set of touchpoints to dynamically monitor and adapt the mediation chains under execution at the architectural level. Touch points can also be used to monitor some aspects of the execution platform like the available services, the protocols used to do so, etc.

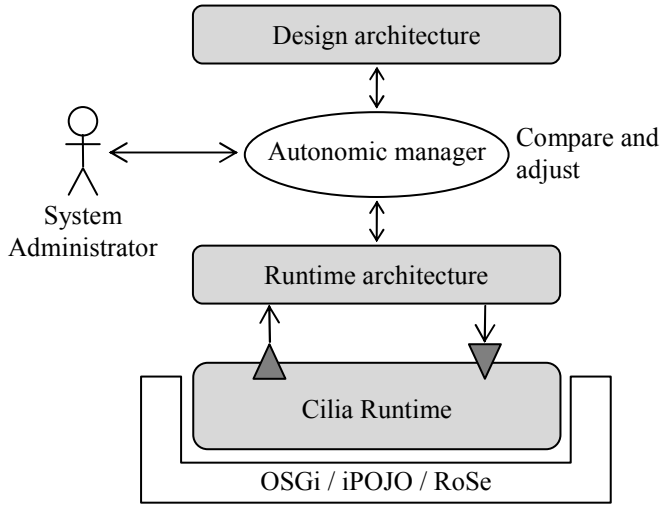

Figure 4. Architecture-based autonomic management.

Monitoring and adaptation features are flexible and configurable. Monitoring, in particular, can be controlled in a dynamic way. This means that Cilia monitoring can be activated or deactivated globally. It also means that the elements to be monitored, and the way they are monitored, can be configured without interruption of service. This allows developers and administrators to use Cilia features in accordance with their needs and objectives. Expectations can obviously vary according to the runtime situation and to the problems that may arise, which is a typical property of administration systems.

Monitoring touch points are used to build the runtime architectures of the mediation chains under execution. The runtime architecture abstracts the adapters, mediators and bindings under execution. It relies on state variables that are used to model the dynamics of the running chains. State variables are attached to global mediation chains but also to their constituents (mediators, adapters and bindings). Their values, called measures, are kept in circular lists in order to keep records of the past. The size of the lists is configurable and can be changed at runtime. Warnings and alarms can be defined on the state variables. When a measure exceeds a 'low' or 'high' threshold, a warning is emitted. When a measure exceeds a 'very low' or 'very high' threshold, an alarm is emitted by the Cilia runtime.

Specifically, the state variables attached to each mediator are the following:

- Scheduler start time,
- Scheduler incoming data,
- Processor start time,
- Processor incoming data,
- Processor outgoing data,
- Processor end time,
- Value of a processor field annotated by the developer,
- Dispatcher start time,
- Dispatcher incoming data,
- Mediator execution time,
- Number of messages sent out by a mediator.

We believe that touchpoints are of utmost importance. They provide ways to access the key elements of a running software system in order to observe it and to change it whenever needed. Specifically, it is possible to dynamically update the following architectural elements:

- A mediation chain can be dynamically added or removed,
- Configuration parameters of a mediation chain can be dynamically updated,
- A mediator can be dynamically removed from a running chain,
- A mediator can be dynamically added,
- A mediator can be dynamically replaced within a running chain (hot swapping),
- Configuration parameters of a mediator can be dynamically updated,
- Configuration parameters of the execution machine can be dynamically updated,
- An adapter can be dynamically replaced. Its configuration parameters can be changed.

B. Architecture-focused tool

We have developed an integrated development and administration environment in Eclipse. This environment, specific to Cilia, is based on the approach presented in the previous section. It allows the modeling of design mediation architectures, with variable aspects, and automatically builds runtime architectures in a configurable way.

A design mediation architecture is then a specification of a chain with some variability. As expressed in the meta-model in figure 5, it is made of abstract and concrete mediators with service dependencies, and bindings with possible cardinalities (adapters are not represented here for the sake of simplicity).

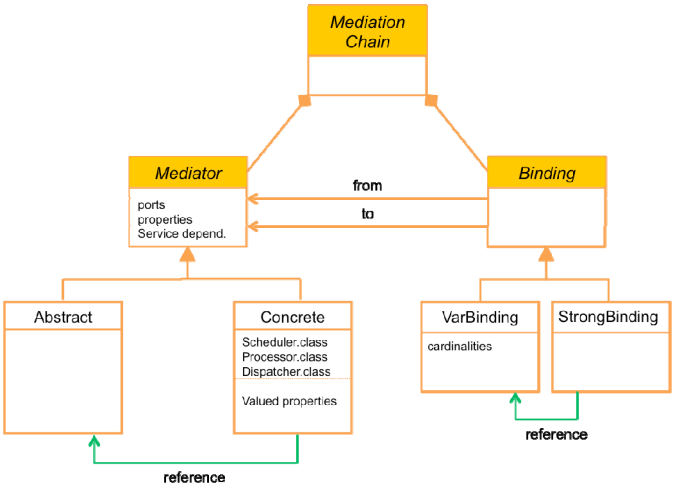

Figure 5. Design architecture meta-model.

Three techniques can then be used to introduce variability in design mediation chains:

- Abstract mediators can be inserted in a chain. An abstract mediator is a high level description, independent of the implementation. This description specifies the mediation operation to be performed and may set some parameters values.
- Cardinality can be expressed in bindings. Cardinalities can be set to (1), (1..*) or (0..*). Thus, a branch of a mediation chain can be optional or multiple. That means that it can be absent or duplicated under various forms, depending on the runtime conditions.
- Service dependencies can be expressed for each mediator. A dependency is expressed with a Java interface and can be characterized by a cardinality and a ranking function. A ranking function can be seen as a utility function, as commonly developed in autonomic computing, used to classify candidate services.

Depending on the target environment, more or less variability can be expressed. The more variability in a chain, the more opportunities for runtime adaptation. But also the more challenging is the task of the administrator (or of the autonomic managers). On the contrary, with little variability in the design, there is no much contingency at runtime for controlled adaptations, which are anticipated at design time (even at a high level of abstraction).

Let us now return to the actimetrics use case introduced in section 2. The purpose of the mediation chain is to collect as much information as possible regarding inhabitant's movements. This information can come from all devices placed in the house, albeit heterogeneous ones.

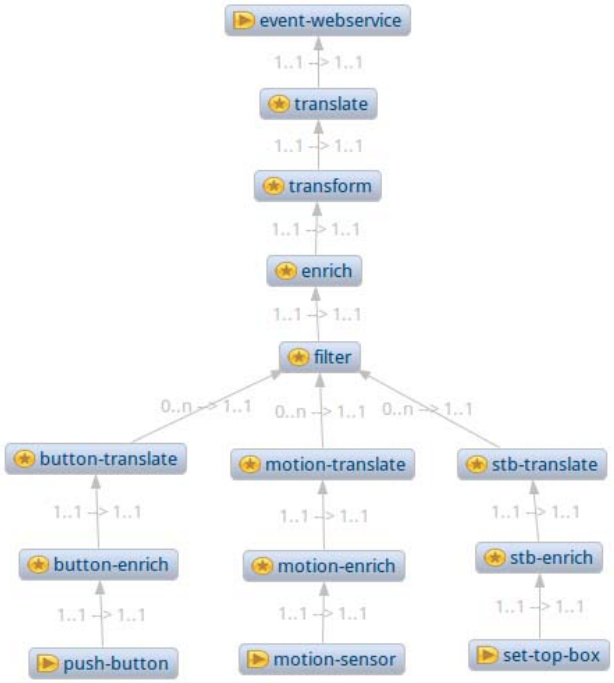

Figure 6. Actimetrics design mediation chain.

As illustrated by figure 6, 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 (timing or location information for instance) 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 into a standard format.
- A branch made of the mediators *set-top-box*, *stb - enrich*, *stb-translate*. This branch gets information from set-top-boxes (stb) when used by the inhabitant, then enriches the captured information and translates it into a standard format.
- 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 (executed on remote server).

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 a base one, meaning that it is always instantiated, regardless of the environment and runtime conditions.

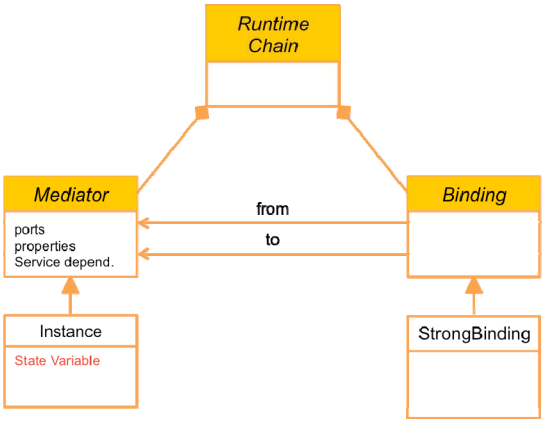

Figure 7. Runtime architecture meta-model.

The runtime architecture, which meta-model is presented by figure 7, stores runtime information about the mediation chains 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. This model is causal in the sense that modifications made on the runtime architecture are reflected on the Cilia runtime; and vice versa. 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. Such approach does not demand to be familiar with the intricacies of Cilia; domain-specific mediation knowledge suffices to manage Cilia-based systems.

Let us get back to the actimetrics use case. An instance of runtime architecture is presented by figure 8. In the figure, state variables are not presented (they are in fact accessible in specific windows of the 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 overseeing the running application.

This allows administrators, human or not, to make better decisions when something unexpected occurs. When the execution context changes (arrival of a new device for instance) or when the quality of service is not sufficient changes can be made while meeting the initial design constraints.

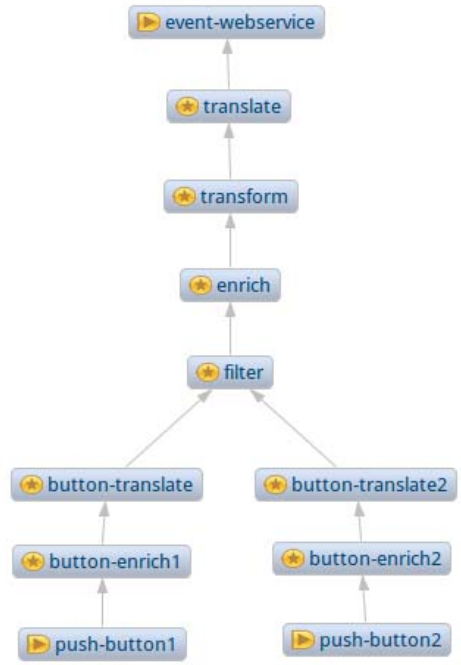

Figure 8. Actimetrics runtime mediation chain.

As shown by figure 9, 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.

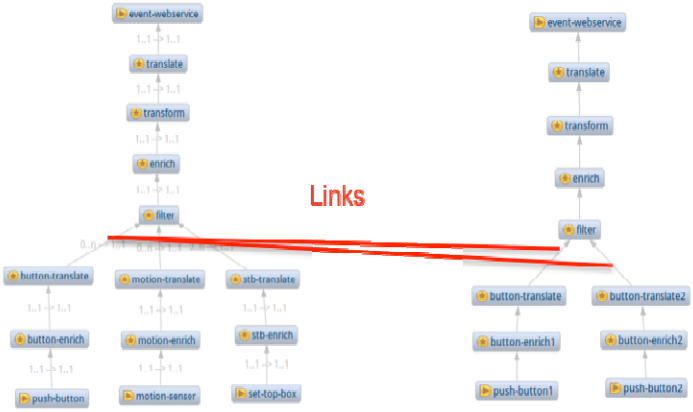

Figure 9. Traceability links between design and runtime mediation chains.

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

Since the 90s, there is a great deal of work on software architectures [24] and Architecture Definition Languages (ADL). Most of these ADLs, which can be seen as Domain Specific Languages, rely on the notions of components, bindings, configurations and constraints. In our architectural languages for design and runtime, we rely on the same concepts.

Runtime architectures are the expression of runtime phenomenon. In that sense, they are models at runtime as introduced in [22,23]. They can be causal, like in our approach, which means that their modifications are reflected on the running artifacts.

In autonomic computing, runtime architectures are used to support analysis and adaptations. An important hypothesis, however, is that architecture correctly mirrors the managed system. Another major assumption regarding causal actions is that there is a correct synchronization between models and running artifacts, which is always a delicate operation.

An important advantage of this approach is that runtime architectures can be used to check that system integrity can be preserved. Indeed, changes are applied to the model first, which allow capturing any violations of constraints or requirements. 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 [7].

For instance, the Acme adaptation framework [8,9] uses a runtime model for monitoring and detecting adaptation needs. Components and connectors can be annotated with properties and constraints to determine when changes are required. An imperative language is used to describe possible adaptations. Similarly in C2/xADL [7,10], the difference between an old architectural model and a new one is computed to create a plan. The plan is analyzed to make sure that the change is valid. The plan is then executed on the running system without restarting it (see [7] for details).

An open research area is the formalization of the target architectural model (design architecture). Some variability has to be introduced in order to make room for runtime adaptation while preserving essential properties and integrity. Variability may concern very different aspects including topology (in terms of components and bindings), security, QoS [25].

Another shortcoming regarding architecture and autonomic computing concerns the lack of tools managing the whole software life cycle. That is, there is a crucial need 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.

Recently, the software engineering community recognized the major role of self-adaptability in the development and maintenance of modern software systems. Many approaches are based today on software architecture and the introduction/specification of a number of control loops to implement such self-management capabilities [18, 19, 20, 21]. Our proposal is in line with these approaches since we try to identify variation points in the design architectures corresponding to control loops in the concrete architectures. We also try to formalize the relationships between these variations points and control loops in order to ease the work of administrators, being human or not.

VI. CONCLUSION

Service-oriented computing allows postponing architectural decisions at runtime, that is to say when the execution conditions are entirely known. This approach is actually used more and more in order to deal with modern software systems that need to be adapted (or to adapt themselves) to meet new requirements or new conditions of the environment in which they are deployed. Such a capability is extremely important in domains like pervasive computing or mobile computing for instance, where execution environments are changing in unpredictable ways.

The downside of this approach is that service-oriented systems are hard to manage. Indeed, the software architecture of those systems is subject to runtime evolutions 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 provide the administrators with a regularly updated representation of the design and runtime architectures and the relationships between them. 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 and runtime architectures. A design architecture defines a solution space and includes some variability regarding the components to be instantiated, the topology, the configuration, the services to be used, etc. The runtime architecture is very different. It models runtime phenomenon with state variables, which are similar to those used in modern control systems [26]. The runtime architecture is used to analyse and adapt the running software whereas the design architecture is used to constraint the adaptation possibilities.

Design and runtime architectures are both based on the notions of abstract components, concrete components, executed components, service dependencies bindings, cardinality. But these notions have different meanings and different attributes that need to be made explicit.

This proposal has been successfully implemented on a use case developed within the Orange Labs in the pervasive health domain. So far, it has been used to help administrators take decisions, at a high level of abstraction, while preserving architectural integrity.

We are now applying this approach to the autonomic management of these very same Orange applications. Indeed, design and runtime architectures provide a way to structure the knowledge of an autonomic manager and to simplify the autonomic reasoning and actions (especially if the architectural model is causal). We are also investigating generic algorithms autonomically adjusting design architectures with variability and runtime architectures, both a deployment time and execution time.

VII. REFERENCES

[1] Weiser, M. 1995. Human-computer interaction. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter The computer for the 21st century, 933–940.

[2] Szyperski, C. 2002. Component Software: Beyond Object-Oriented Programming 2 Ed. Component Software Series. Addison-Wesley Professional.

[3] Papazoglou, M. P. 2003. Service-Oriented Computing: Concepts, Characteristics and Directions. In WISE'03: Proceedings of the Fourth International Conference on Web Information Systems Engineering, Los Alamitos, CA, USA, 3–12.

[4] Cervantes, H. and Hall, R. S. 2004. Autonomous Adaptation to Dynamic Availability Using a Service- Oriented Component Model. In ICSE'04: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, Los Alamitos, CA, USA, 614–623.

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

[6] P. Lalanda, J. McCann and A. Diaconescu, "Autonomic Computing : Principles, Design and Implementation", Springer Verlag, 2013.

[7] 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.

[8] 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.

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

[10] 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.

[11] Krueger, Charles W "Variation Management for Software Product Lines". The Second Product Line Conference (SPLC2), San Diego, CA, USA, 2002.

[12] J. van Gurp, M. Svahnberg, J. Bosch, "On the Notion of Variability in Software Product Line", Proceedings of the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, pp.45-54, 2001.

[13] Kang, K.C, Cohen, S., Hess, J., Nowak, W., and Peterson, S. "Feature oriented domain analysis feasibility study". Technical Report CMU/SEI-90-TR-21, Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University, 1990.

[14] M.Antkiewicz, K.Czarnechi "FeaturePlugin:Feature Modeling Plug-in For Eclipse" OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

[15] Pure-systems GmbH. Variant Management with Pure::Consul. Technical White Paper. Available fromhttp://web.pure-systems.com, 2003.

[16] M. Satyanarayanan, "Fundamental challenges in mobile computing," in the Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, New York, USA, 1996, pp. 1–7.

[17] Escoffier, C., Hall, R.S., Lalanda, P., "iPOJO: an Extensible Service-Oriented Component Framework". In: IEEE International Conference on Services Computing, Los Alamitos, CA, USA, IEEE Computer Society (July 2007) 474_481.

[18] Y. Brun *et al*, "Engineering self-adaptive systems through feedback loops", In : Self-adapting systems, LNCS 5525, pp. 48-70, 2009.

[19] B. Cheng et al, "Software engineering for self-adaptive systems : a research roadmap", In : Self-adapting systems, LNCS 5525, pp. 48-70, 2009.

[20] S.W. Cheng *et al*, "Improving architecture-based self-adaptation trhough resource prediction", In : Self-adapting systems, LNCS 5525, pp. 48-70, 2009.

[21] O. Nierstrasz, M. Denker and L. Renglli, "Model-centric, context-aware software adaptation", In : Self-adapting systems, LNCS 5525, pp. 48-70, 2009.

[22] R France, B Rumpe, "Model-driven development of complex software: a research roadmap", In : Future of Software Engineering, 37-54, 2007.

[23] G Blair, N Bencomo, RB France, "Model@runtime", Computer 42 (10), 22-27.

[24] M. Shaw and D. Garlan, "Software architecture: perspectives on an emerging discipline", Prentice Hall, 1996.

[25] G. Tamura, "QoS-CARE: A Reliable System for Preserving QoS Contracts through Dynamic Reconfiguration", PhD thesis, 2012.

[26] K. Ogata, "Modern control engineering" (5th edition), Boston, MA, Prentice Hall, 2010.