Autonomic Enterprise Service Bus

Denis Morand, Isaac Noé Garcia Garza, Philippe Lalanda

Autonomic Enterprise Service Bus

Denis Morand
Schneider Electric
220 rue de la chimie
38 000 Grenoble
France
Denis.Morand@imag.fr

Issac Garcia
Grenoble University
220 rue de la chimie
38 000 Grenoble
France
Issac-Noe.Garcia-Garza@imag.fr

Philippe Lalanda
Grenoble University
220 rue de la chimie
38 000 Grenoble
France
Philippe.Lalanda@imag.fr

ABSTRACT
In this paper, we describe the work that has been realized in order to add autonomic features to Cilia, an open source mediation framework developed jointly by the LIG/Adele team at Grenoble University and Orange Labs. Cilia has been designed for data and application mediation and is used in several industrial use cases. This paper, in particular, develops the notion of state variables and action variables adapted from control theory. It shows how they can be used to follow the state of integration chains and to bring changes at runtime.

Keywords
Enterprise Service Bus, Autonomic computing.

1. INTRODUCTION
Service-oriented Computing (SOC) has recently emerged in the software engineering community [1, 2, 3]. The very purpose of this reuse-based approach is to build applications through the late composition of independent software elements, called services. Services are described and published by service providers; they are chosen and invoked by service consumers at runtime. This is achieved within a service-oriented architecture (SOA), providing the supporting mechanisms.

Service orientation brings in major software qualities. First, it favors the rapid development of quality software applications. It also promotes weak coupling between consumers and providers, reducing dependencies among composition units. Finally, late binding and substitutability improve adaptability. Since a service can be chosen or replaced at runtime, it is easier to improve the way requirements are met.

A number of implementations have been proposed in the last years. Web Services (www.w3c.org), for instance, represent a solution of choice for software integration. UPnP (www.upnp.org) and DPWS (Devices Profile for Web Services) are heavily used in pervasive applications in order to implement volatile devices. OSGI (www.osgi.org) and iPOJO (www.ipojo.org) provide advanced dynamic features to many software systems.

That being said, service-oriented computing also suffers from important limitations. In particular, it is complex to conceive and implement an application made of dynamic, heterogeneous services and required to meet non functional requirements. Doing so requires deep expertise. Cross-technology applications require almost unavailable skills. In addition, as of today's state-of-the-art, service composition cannot be based only upon service specifications. Syntactic compatibility does not ensure semantic compatibility. In practice, service composition is based on unexpressed assumptions and rules allowing reaching the expected results. A composition of services has also to reach a set of pre-defined non functional qualities (like security for instance) which requires the production of complex, often non flexible code. In the general case, such code cannot be automatically generated at composition time.

We believe that without effective solutions for easy and correct service composition, SOC orientation will be limited to narrow, very specific domains of applications. In this paper, we present a mediation tool allowing the effective integration of services, called CILIA. This tool, based on a domain-specific component model, allows the creation of mediation chains implementing the necessary non functional operations when calling a service. It has been successfully used in several use cases, at France Telecom in particular. It however appears that the management of such tools is difficult in the sense that it has to deal with the high volatility of services. The provisioning of high-level services based on heterogeneous, distributed and mobile software applications and hardware devices is a difficult task. Dynamism is a particularly complex and remains an important issue in service-oriented computing. This is required as applications evolve with their execution contexts, when software and hardware components get modified, or as users change their computing environments or desires.

Autonomic Computing promises a solution to the aforementioned problem, by endowing software systems with self-management capabilities that would minimize or eliminate the need for human intervention [9,10]. If successfully implemented, autonomic pervasive applications would inherently feature critical properties such as safety (including fault-tolerance and security) and self-adaptation to internal and external changes (including self-configuration, self-optimization and self-repair). However, building autonomic properties into pervasive systems remains a difficult task. Reusable solutions for the development of Autonomic Management (AM) systems remain rather limited and generic. There is however a stringent need for more specific, readily-usable frameworks for facilitating the development of AM solutions for different computing domains

In this paper, we also examine how Cilia has been made autonomic. The paper is organized as follows. First, background about Enterprise Service Buses is given. Then, the CILIA component model and associated runtime framework is presented. The fourth section is concerned with autonomic extensions brought to Cilia. More precisely, the notion of state variable is presented. Approach used for Cilia introspection and adaptability is also presented. The paper ends with concluding remarks.

2. ENTERPRISE SERVICE BUS

The activity of integrating disparate information sources in a timely fashion is known under the name of mediation. Mediation has been historically used to integrate data stored in IT resources like databases, knowledge bases, file systems, digital libraries or electronic mail systems [4,5,6]. It is now also used to allow interoperation between heterogeneous software applications. In this context, mediation software stands between client applications and provider applications. Its purpose is to enable a consumer to easily and properly use a provided service. We use the term mediation service to refer to software allowing the integration of service-based applications.

Service mediation implements all the operations that are necessary to enable the actual communication between a set of service-based applications. The most common functions to be provided are the following:

- Communication. The primary purpose of mediation is to enable applications using different communication protocols to interoperate. This is implemented by means of protocol transformations as in a network bridge. This function can also play the role of a broker, hiding for instance the applications network addresses
- Syntactic alignment. The purpose of this function is to align data formats. This can be done between each application or through an intermediary format. In the latter case, the number of data transformations to be made is obviously reduced.
- Semantic alignment. The purpose of this function is to align data semantics. In the absence of recognized and used standards, applications develop different ontologies to represent (static and dynamic) knowledge. Automating ontologies alignment is a major research challenge of the service community.
- Non-functional properties. The purpose of this function is to ensure certain quality properties in the application exchanges, like for instance security or availability.
- Persistency. The purpose here is to keep track of all exchanges between applications. The mediation layer can provide logging support for all requests, responses and data.
- Monitoring. The purpose of this function is to collect data for monitoring systems that verify that the expected quality of service is being achieved.
- Business logic code. The mediation layer can be used to insert business logic code, like an access to a database for instance. Even though this approach can be particularly useful, its use is rather not recommended. It actually introduces confusion as it produces an architecture where the business logic code and technical code are mixed.

Encapsulating such operations in dedicated software is clearly good software engineering practice. Mediation software provides a single point of interface to the different applications implied in the communication. This reduces the number of connections needed and facilitates change management. Mediation also provides an isolation layer from software details and, if appropriately configurable, permits the quick and cost-effective development of new services. The mediator layer improves reusability and evolution of applications. It also permits the transparent addition of new QoS properties such as security, reliability, etc. Finally, it leads to the improvement of the scalability of the whole system.

The mediation layer is often achieved through the use of an EAI (Enterprise Application Integration). EAIs usually appear as monolithic software based on the *hub and spoke* pattern. EAIs have been widely used in the last few years. They now must face heavy criticism, due to their cost and size. We believe that this is partly due to the fact that EAI have gone too large, exceeding their initial functional scope. Also, a single EAI is often used to integrate all applications of a company. Any new service using existing applications has to go through such unique EAI.

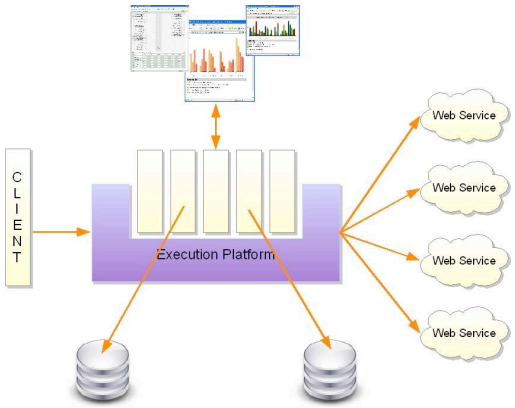

Figure 1. An ESB provides a run-time environment mediating Web service applications.

The emergence of service-oriented computing has fostered architectural evolutions. In particular, Web services aim for lighter integration solutions and have initiated the definition of Enterprise Service Buses, or ESB [7]. An ESB is a communication bus located between clients and Web Services and hosting potentially distributed mediation operations. Mediation is frequently organized as mediations chains that transport the request from the client application to the service provider and the answer the other way around. The mediation chains can be decomposed into light weight components called mediators that implement simple operations. ESsB provide a unique interface to all applications and eliminate all direct contact between applications, as all communication is made through the bus.

ESBs raise major design and implementation-related issues. They have to meet stringent requirements, including:

- **Lightweight**. The primary demand to be satisfied by an ESB is to be lean. The purpose of companies we are working with is to install ESBs on demand to make targeted applications interoperate. They want to avoid the EAI syndrome where all possible applications are linked to a single, fat EAI.
- **Efficient**. This is a major feature for all integration software. Time needed to align data and to perform non functional operations should not impact the quality of the overall service provided by the integration.
- **Easy to install and manage**. Since ESBs are to be used frequently, on the creation of a new service, their installation, configuration, management have to be simple.
- **Flexible**. Non functional requirements evolve over time. Then, it should be easy to modify or add mediation operations in order to adapt at run time the way applications are integrated.
- **Easy to use**. Programming and run-time models have to be simple. Once again, the point is to avoid getting back to nasty solutions where programming and maintaining mediation code is so hard that consultancy or dedicated teams, expensive in both case, is unavoidable.

- **Error handling**. One of the salient requirements brought by our industrial partners is the ability to easily deal with errors. Application integration is subject to many errors (communication failures, inadequate or non running services, incorrect answers, etc.). An important part of integration code is actually dedicated to error handling.

Several solutions have been recently proposed. We can actually distinguish between two architectural approaches. The first approach is to extend a J2EE server. This solution consists in developing an ESB on top of an existing J2EE application server (WebSphere ESB or SpringIntegration for instance). The first appeal of this approach is obviously to reuse the J2EE programming model and the servers facilities. The result is however often very big in size. In addition, the programming model is not perfectly adapted to mediation. As a remedy, domain-specific tools like Camel[1] have been proposed.

A second approach is to develop dedicated tools. Many tools have been proposed in recent year like Codehaus Mule[2]. A standard, called JBI for Java Business Integration, has actually been proposed to structured ESB. JBI is based on the JSR 208 that standardizes a component-based architecture. Components are simple objects orchestrated by a controller named Normalized Message Router. A distinction is made between pure mediation component (Service Engines) and components used to interact with other resources (Binding Components). Some open source tools based on JBI are emerging like Apache ServiceMix[3] or ObjectWeb Petals[4].

But, in all cases, the mediation solutions are very technical and technology-driven. Mediation chains are hard to build, deploy and maintain. They are also uneasy to change and to reuse. Most ESBs mentioned here have been tested by our partner in real-size industrial use cases and failed to meet the requirements presented here before. We believe that there is a clear need to focus on mediation operations, to consider them as first order objects and to treat them accordingly. Complex, low-level technical details should be hidden by a mediation tool in order to allow developers to focus on the integration of heterogeneous applications.

3. CILIA

In this section we present an integration component model, called Cilia [8], which addresses the interoperability issues between heterogeneous data sources (e.g. applications, devices, etc.) and targeting systems. Cilia is based on a component model approach which emphasizes the reuse and separation of concerns. A mediation application in Cilia is a set of component instances interacting in a loosely coupled way through, but not limited to, event-based protocols.

Cilia takes the form of a domain-specific component model. As a reminder, such a model defines a language to specify components, a language to assemble these components and an execution framework. Cilia naturally includes these three elements.

In Cilia, components are called mediators whereas components assemblies are called mediation chains. They can be both defined in a specification file with an XML syntax. Components are defined by input and output ports, by some properties and by three Java classes, namely:

- **A scheduler class**. The purpose of this constituent is to synchronize data reception. It intercepts incoming data (requests), store them and launch their processing. The processing decision can be time-based, content-based or, any other condition on relation with the mediation context (e.g., waiting all needed data). For instance, a *periodic* scheduler triggers the processing with the collected data periodically, a *correlation* scheduler waits for all the correlated messages to trigger the processing, and an *immediate* scheduler triggers processing upon data arrival.
- **A processor class**. The processor performs the mediation algorithm *per se*. When notified by the scheduler, it processes the collected data and passes them to the dispatcher. *StringSplitter* is an example of processor that splits the received data using a regular expression. *StringAggregator* in another example that builds a new data concatenating the received ones.
- **A dispatcher class**. The dispatcher receives the processed data from the processor. This constituent decides on the data destination and triggers their delivery. The dispatcher choice is a logical destination because of loosely coupled relations between mediators. The *Multicast* Dispatcher is an example where processed data are delivered to all the connected components. The *content-based* dispatcher delivers the processed data to the chosen destination based in the data content.

Software developers concentrate on the Processor class where they express the mediation logics. The Scheduler and Dispatcher classes can also be developed but, generally, are simply reused. Reference implementations have been provided in order to quickly implement the most common Enterprise Integration Patterns (EIP). For instance, several Schedulers have been implemented to deal with most time-based processing conditions. In the case where a scheduler (or a dispatcher) has to be developed, high level APIs are provided. Thus, many technical details like the way data are stored for instance are not handled by a domain programmer.

Ports are typed. Mediators are connected through compatible ports, using the Cilia assembly language, in order to form mediation chains. Binding are based on the two following elements associated with mediators:

- **Collector**: The collector is the binding constituent which implements the communication protocol to receive data. The data could be received from other mediator or from external communication protocol or application.
- **Sender**: The sender is the binding constituent which implements the communication protocol to send the resulting data. This data could be sent to another mediator, could also be sent using some standard communication protocols, or sent to another application. This component is associated to the dispatcher mediator instance in execution.

A binding specification describes how communication is established. That is, a binding specifies which collector and sender are used for the communication between two mediators and how they need to be configured in order to assure correct communication. For a binding specification which uses a topic-based system, the binding should known which collector/sender it needs to use and how to set-up them with the correct topics when adding them to the corresponding mediators.

[1] http://camel.apache.org/

[2] http://mule.mulesource.org/

[3] http://servicemix.org/site/

[4] http://wiki.petals.objectweb.org/

Bindings specifications are independent of mediators logic, thus mediators could use any binding specification. There are three binding types. The first binding type describes how to communicate between two mediators, thus a sender and collector must respectively be declared for the receiving mediator and the transmitting mediator. The second binding type is the one that allows mediators to receive data from an external system, e.g. a database or through a communication protocol. Therefore, only a collector is defined. The third binding type, is the one that is used to deliver data to an external service or application, thus, only a sender is defined.

As illustrated by Figure 4, Cilia is entirely built on top of OSGI and iPOJO, the Apache service-oriented component model. OSGi provides the base mechanisms for modularity and dynamicity. iPOJO is the Apache service-oriented component model. It facilitates the development of dynamic component-based applications on top of OSGI through, in particular, the autonomic management of service dependencies.

The Cilia framework handles the execution of the mediation chains. Technically speaking, it builds and runs a number of iPOJO components realizing the different functions presented before with the expected quality of service.

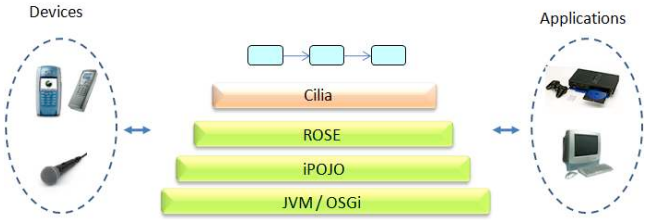

Figure 2. Implementation stack

This platform also integrates a specific module, called ROSE, whose purpose is to constantly reflect the state of the computing environment in the execution machine. ROSE [12] captures services in the computing environment and reifies them as iPOJO components (proxies) in a local registry. It then manages the proxies lifecyle, creating or deleting them as a reaction to arrival and departure of matching services. ROSE also provides advanced facilities to specify the services of interest according to different properties like type, protocol, or number. ROSE currently handles a number of protocols, including Web Service, DPWS, UPnP, Zigbee and Bluetooth. It is available on ObjectWeb (wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose).

Finally, let us note that a number of common mediators and bindings have been defined in order to implement the most usual EIP (Enterprise Integration pattern). These mediators can be directly inserted in a given mediation chain, with an appropriate configuration. The integration designer specifies thus declare a integration chain with base mediators, configuring them and binding them through ports.

CILIA is currently made available as an open source framework (http://wikiadele.imag.fr/index.php/Cilia). It is experimented by our industrial partners, in particular France Telecom and Schneider Electric, on concrete use cases pertaining to the IT domain and pervasive domain.

4. AUTONOMIC CILIA

4.1 Monitoring

Making Cilia (and ESBs in general) autonomic is an essential requirement to deal with dynamic environments, which is the case for most service-based settings. Autonomic systems are made of managed artifacts and one or several autonomic managers. Managed artifacts are the software entities that are automatically administered by the system. Here the managed artifacts are clearly the mediation chains. The autonomic manager is the module in charge with the run time administration of the managed artifacts. The purpose of the autonomic manager, in our case, is to create and adapt the integration process, using the dynamic capabilities of the underlying component model (Cilia). It is driven in its decisions by high level goals set by administrators.

In this paper, we focus on the managed artifacts (Cilia mediation chains), not on the autonomic manager. Our purpose here is actually to show how the Cilia framework has been evolved in order to be appropriately introspectable and adaptable. Of course, these two features constitute the necessary supporting mechanisms to conduct autonomic reasoning (through the implementation of a MAPE-K loop at the autonomic managers' level). It also turns out that it represents a big part of the work to be realized in order to make an already existing system autonomic!

So, in order to enhance Cilia, we successively realized the following tasks:

- Identify the internal aspects of Cilia that are to be monitored at runtime and under which conditions,
- Capture these aspects in a timely fashion and present them in an appropriate form,
- Identify the internal aspects of Cilia that are to be changed at runtime,
- Provide the internal mechanisms allowing the dynamic adaptation of these changeable aspects,
- Provide the adaptation APIs allowing an autonomic manager to actually trigger modifications.

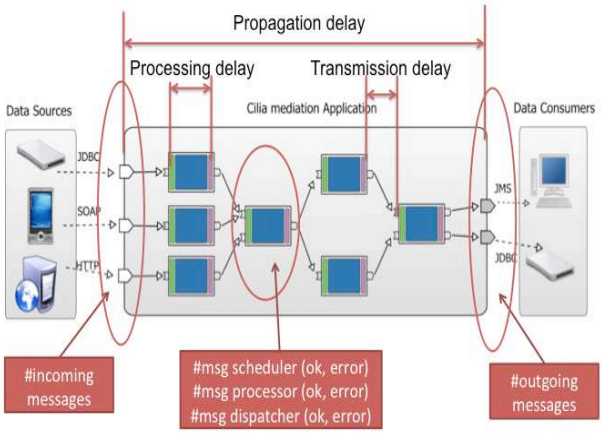

Figure 3. Aspects to be monitored.

We realized that many aspects of the Cilia framework had to be monitored. Some of them directly relate to the execution machine like for instance CPU, memory, threads, base OSGi feature, etc. Other aspects are concerned with the Cilia internal mechanisms supporting the execution of mediators and mediation chains.

Examples of essential aspects to be monitored are illustrated by figure 3. They include:

- The running mediators and their bindings (this is the runtime architecture of the mediation chain)

- The number of incoming messages per port and for each mediator,
- The number of outgoing messages per port and per mediator,
- The size of the different messages,
- The number of calls to the processor for each mediator and the associated performances,
- The mean processing time for each mediator,
- The time needed to traverse a mediation chain (propagation delay),
- The time of communication between two Cilia mediators (transmission delay),
- The latency time for each Cilia mediator (transmission delay).

Collected data are to be used in a "classical" autonomic loop in order to adapt the mediation chain or the mediators' configuration. Actions that can be undergone currently are much related to the way the Cilia framework is implemented. For instance, the size of different buffers (storing input and output messages) can be modified at runtime. Similarly, the number of threads used to manage messages can be adapted.

4.2 Knowledge representation

A major issue in autonomic computing is to represent knowledge. Such knowledge is domain-specific and system-specific. In our case, it relates to the domain of component-based integration and to the Cilia framework. It has however to rely on a generic formalism in order to allow high level reasoning. It is also a necessity in order to maintain a clear separation between domain knowledge and autonomic managers.

To make Cilia autonomic, we took inspiration in control theory. More precisely, we introduced the notions of state and action variables to respectively monitor and act on Cilia:

- **State variables** are used to quantify the stability of the integration process over time. State variables are set, and updated, by monitoring functions.
- **Action variables** are used to modify the integration process. Action variables are set, and updated, by the action part of the autonomic manager.

State variable and action data are different; even if they can be the same in some situations. This is an interesting finding in the sense that different code insertions are needed in Cilia. Some code is needed to get state variables; some code is needed to set action variables. Common APIs are not desirable since they would be completely artificial in many cases.

A state variable is a data that quantifies an important aspect of a supervised process. For instance, it can be the size of a buffer or the number of running threads in Java. A state variable is a numeric value that comes with a validity interval that is used to specify a viability zone for a process. A set of state variables is a set of such variables. It is used to define a viability zone for a process. This means that when all the variables in the set are all in a well-defined interval, then the process is executing correctly.

An action variable corresponds to a data related to the supervised process that can be changed. It can be, for instance, the size of a buffer. An action variable can be directly related to a state variable, but it is not mandatory. For instance, the number of threads can be a state variable and not an action variable (it cannot be externally modified).

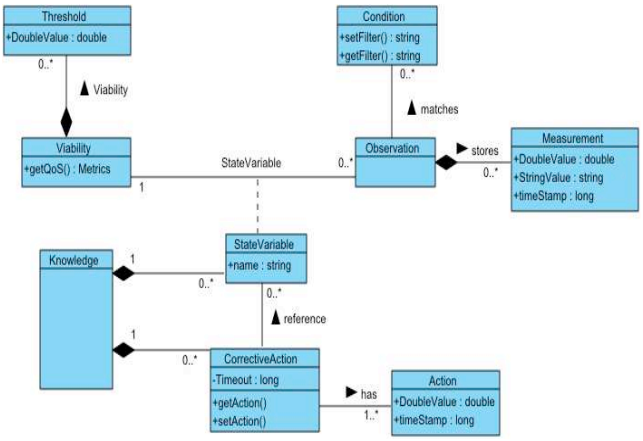

Figure 4. Metamodel state and action variables.

Figure 4 presents the meta-model that has been defined in Cilia for the state variables. It is interesting to note that state variables are related to observations that are created only when a given condition is true. This is a way to limit the number of stored values for state variables. Only data related to an abnormal situation are retained. A similar meta-model has been defined to specify action variables. There is an optional link between the two meta-models since state variables can be directly linked to action variables in some cases.

State and action variables are used to express all the relevant information of the Cilia framework, being at the mediators' level or the mediation chains' level.

5. IMPLEMENTATION

Let us now detail, in this section, how state variables are acquired and how action variables are trigerred. This is done by two important activities, defined in the IBM MAPE-K loop [9], namely monitoring and acting:

- Monitoring relies on the introspection capabilities of the Cilia framework. It provides the timely elements (state variables) necessary to understand runtime mechanisms and to analyze aspects like performance, reliability, etc.
- Acting relies on the dynamic adaptation capabilities of Cilia. It triggers operations related to the chains and mediators lifecycle, like create, replace, change a mediator. It also triggers changes on more general action variables, as introduced in the previous section.

5.1 Monitoring

Monitoring in Cilia has been designed in order to meet three major principles:

- Monitoring actions must not affect performances. They must not reduce the global performance of the mediation chain and the framework platform,
- Monitoring should be configurable at runtime. Depending on the execution situation, different pieces of information are necessary.
- Monitoring should be based on several points of view whenever possible. Introspection is complete when observing two independent points (often related to state variables).

These three base requirements are not incompatible. At anytime, only the needed aspects are monitored in a redundant fashion. This is a way to save performances of the global framework while presenting a faithful vision of runtime situation.

To meet the first two requirements, we have implemented monitoring capabilities as pluggable functions. These functions can be added at different places of the framework in order to conform to the third requirement (several points of view).

As previously said, Cilia is entirely built on top of OSGi/iPOJO. Let us recall here that iPOJO is a Java-based, dynamic component framework. One of the main goals of iPOJO is to make developing dynamic applications as simple as possible. To this end, the overall approach is to keep a component as close to a "plain old Java object" (POJO) as possible. The code of a component should focus on business logic, not on mechanisms for dynamism or other non-functional requirements. iPOJO provides an extensible component container that manages all issues regarding dynamism and can be extended to support other non-functional concerns, such as configuration, persistence, and security. The POJO component is connected to iPOJO by configuring the component instance container, which consists of declaring component type meta-data that will be used by the container for run-time management. IPOJO containers are not monolithic, but are composed of *handlers*, as illustrated by figure below. Each handler manages a non-functional concern. Handlers are plugged into the component instance container at run time. Only the required handlers are plugged into the container. The resulting container manages the interaction between the POJO and the external world. Custom handlers can be developed for iPOJO, allowing developers to handle other non-functional concerns. Handlers perform their function by monitoring POJO component member fields to inject new member field values or be notified of a value change. Additionally, handlers participate in the life-cycle management of component instances. In iPOJO, a component instance is either valid or invalid. An invalid instance cannot be used by anyone, until it becomes valid. An instance is valid if and only if all plugged in handlers are valid.

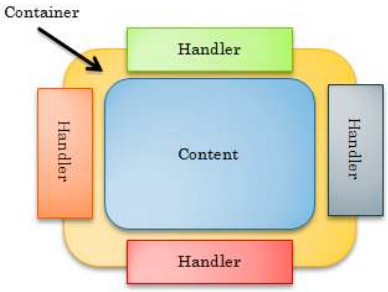

Figure 5. An iPOJO instance and its container

Monitoring functions are implemented using iPOJO features:

- iPOJO components are used for the global monitoring functions. Here, we refer to monitoring functions related to the execution machine and to the integration chains,
- iPOJO handlers are used for the fine-grained monitoring of mediators.

Thanks to iPOJO dynamic capabilities, all the monitoring functions can be added, replaced, removed or re-configured at runtime. Thus, monitoring can be tuned at runtime in a context-aware fashion. Such tuning can be done, through specific APIs, by an administrator or by autonomic managers.

Figure 6 presents the introspection architectural design. It shows that, for each mediator, implemented as an iPOJO component, a *MediatorMonitorHandler* can be defined. This handler presents an event-based interface providing publish/subscribe interfaces to potential clients (called *ApplicationMonitoringImp* in the figure).

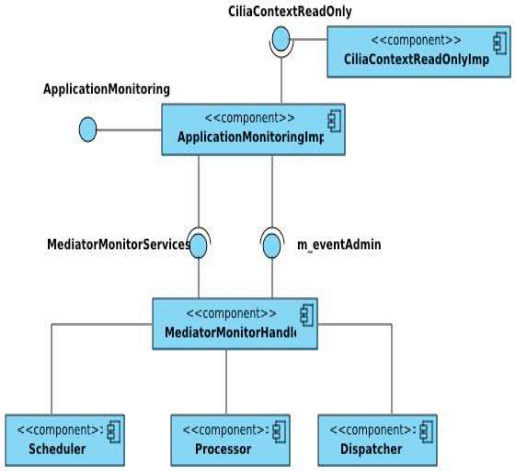

Figure 6. Introspection design

The *MediatorMonitorHandler* implements the introspection part of a mediator. The handler has been designed not to be intrusive. In particular, it is dynamically pluggable and configurable. Data to be collected and associated periods of collect can be adapted. The *MediatorMonitorHandler* collects events regularly fired by the Scheduler, Processor and Dispatcher entities. State variable values are created, time-stamped and sent to the clients based on the publish/subscribe mechanism. In this case, the *MediatorMonitorHandler* acts as a publisher. Communication between the handler and the clients is asynchronous, here again in order not to affect global performances.

Figure 7 shows an excerpt of the APIs allowing the definition of data to be surveyed and the associated conditions.

```
public interface StateVariableSet {
    /* state variable management */
    public void add(String stateVarId, int window);
    public void remove(String stateVarId);
    public void enable(String stateVarId) ;
    public void disable(String stateVarId);
    public String[] keysStateVariable();

    /* Condition to fire value */
    public boolean setCondition(String stateVarId, String ldapFilter);
    public String getCondition(String stateVarId);

    /* Observation stored ( measures) */
    public void addMeasure(String stateVarId, Object m);
    public Object[] getMeasures(String stateVarId);
    public void clearMeasures(String stateVarId);
}
```

Figure 7. Excerpt of provided interfaces

Events provided by the mediator handlers allow the runtime re-construction of the integration chains. More precisely, our system permits:

- The discovery and presentation of all the running chains,
- The discovery of all the running mediators that can be monitored (discovery can be impeached by specific configuration),
- The presentation of the availability of mediators and chains,
- The analysis/profiling of the internal data (state variables) of each monitored mediator,

- The capability to change the state variables to be monitored and or to change the monitoring conditions of such state variables. Changing these monitoring conditions can increase or reduce the sampling rate (the condition act as the control flow of measurement),
- The tracking of any modifications observed on the topology, as well as modifications on internal data and state of mediators.

To summarize, the purpose of the monitoring capabilities in Cilia is to build and maintain a (runtime) model of the integration chains and of the constituting mediators. Data structures storing state variables are built for each mediator. Thus, a current vision is presented and an historic is maintained.

This is illustrated here after by figure 8 which provides a global view of the monitored data.

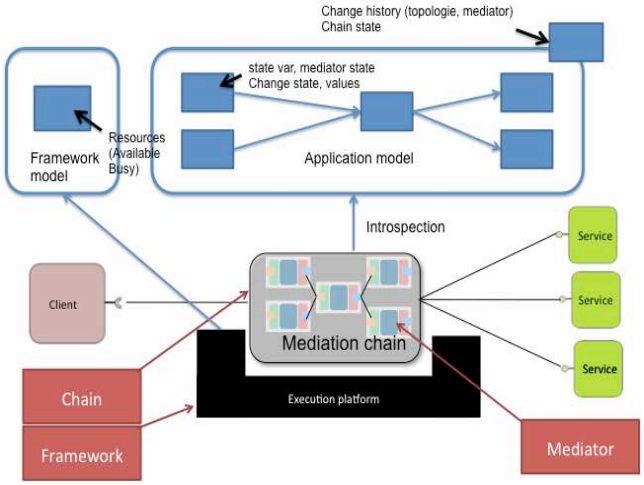

Figure 8. Introspection overview

5.2 Acting

Adaptation capabilities in Cilia have been designed in order to allow fine-grained and large-grained runtime modifications. More precisely, the following operations are possible:

- Create or delete a complete chain,
- Add a new mediator in a running chain,
- Replace a mediator in a running chain,
- Create a binding between two mediators,
- Modify a binding between running mediators,
- Change an attribute of a running mediator. An attribute can control a very precise aspect of a mediator like, for instance, the size of the message buffer,
- Lock / unlock message flow (in order to reach a quiet state needed to perform a change),
- Etc.

As for monitoring functions, adaptation capabilities rely on iPOJO. They are implemented as iPOJO components for adaptations of the execution machine or of mediation chains. They are implemented as iPOJO components handlers for mediator level actions. Similarly to monitoring, components and handlers are pluggable, changeable and removable on demand. Figure 9 presents the architectural design. It shows that, for each mediator, a *MediatorActuatorHandler* can be defined. This handler presents an interface essentially made of effectors.

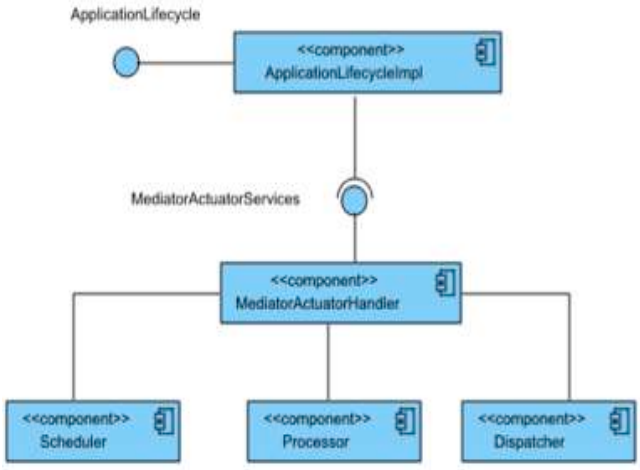

Figure 9. Effectors design

One of the main issues to support adaptation is of course to maintain the coherency of the integration chains, *i.e.* messages and data stored in buffers have to be preserved (knowing that the execution machine cannot be stopped/restarted).

We have therefore defined three states for a mediator: running, stopped and quiescent. When a running mediator is locked, it passes to stopped. When a stopped mediator is empty, it passes in the quiescence state. Finally, when a quiescent mediator is unlock, it passes to running.

Modifying a mediator or a chain is then done according to the following process (realized by the framework itself):

- Lock the running mediator to be changed; all incoming messages are intercepted and stored,
- Connect the new mediator when the mediator to be changed is in the quiescent mode,
- Inject stored messages in the new mediator,
- Remove the locked mediator.

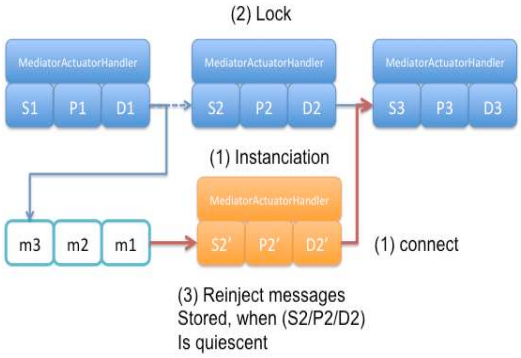

Figure 10. Effectors design

This process is illustrated by figure 10 where (m1, m2, m3) are messages, S stands for Scheduler, P stands for Processor and D for Dispatcher.

6. CONCLUSION

Cilia is a domain-specific component model dedicated to mediation. It is built on top of service-oriented technology [3], especially OSGi and iPOJO. It however shows important management complexity and autonomic solutions are much

required today to be used in industry. This is especially true when it comes to pervasive environments that are characterized by their high dynamism.

The purpose of the autonomic computing initiative [9,10] is to limit the need for human intervention in computer management processes by enriching software applications with self-management capabilities, such as self-building, self-optimization, self-configuration, self-repair and self-protection. Conforming to this approach, autonomic management abilities would enable software applications to seamlessly and transparently self-adapt to their changing environments and evolving business goals. Therefore, autonomic computing seems to provide a viable solution to the difficult concerns of runtime adaptation of integration tools.

According to IBM blueprint architecture, autonomic systems are made of managed artifacts and of autonomic managers. Managed artifacts are the mediation chains. Autonomic managers are the decision-makers. In our case, the purpose of an autonomic manager would be to create and adapt the integration chains using the dynamic capabilities of the underlying component model (that is Cilia).

The purpose of this paper was to show Cilia has been enhanced to be truly adaptable and, then, "autonomic-ready". Cilia has been extended with introspection capabilities and adaptation APIs. To do so, we have defined the notion of state variables and action variables, inspired from works in control theory [15]. Control theory is a cross-disciplinary domain that deals with the behavior of dynamical systems. State variables allow to follow low level programming aspects belonging to the Cilia core framework and to adjust them whenever necessary through the use of action variables. Adaptation are safe and controlled; running mediators can only be changed when they are in a quiescence state. This guarantees that no data is lost upon a modification.

We are now investigating the design and implementation of the autonomic managers since the managed artifacts can effectively be managed now! Our approach here is to express reference integration chains that can be used as guidance to perform adaptations. Reference integration chains will be expressed in conformance with a dedicated meta-model. These models will include explicit variability making room for run time decisions [11].

7. REFERENCES

[1] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented computing: Introduction. Commununications of the ACM, 46 (10):24–28, October 2003

[2] SECSE team, "Toward service-centric system engineering", ICSOC, Trento, Italy, 2003.

[3] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. IEEE International Conference on Services Computing (SCC), pages 474–481, 2007.

[4] G. Wiederhold, "Mediators in the Architecture of Future Information Systems," Computer, vol. 25, no. 3, 1992, pp. 38–49

[5] G. Wiederhold and M. Genesereth, "The Conceptual Basis for Mediation Services," IEEE Expert, vol. 12, no. 5, 1997, pp. 38–47

[6] P. Lalanda, L. Bellissard and R. Balter, " Asynchronous Mediation for Integrating Business and operational Processes," IEEE Internet Computing, vol. 10, no. 1, 2006, pp. 56–64

[7] C. Herault, G. Thomas, P. Lalanda, "A service oriented mediation tool" in Proceedings of the 4th IEEE International Conference on Services Computing (SCC'07), 2007, Salt Lake City, USA

[8] Garcia, Pedraza, Debabbi, Lalanda, Hamon, "Towards a service mediation framework for dynamic applications", IEEE APSCC, 6-10 december, 2010, Hangzhou, China

[9] D. M. Kephart, Jeffrey O. et Chess. The vision of autonomic computing. Computer, 36, 2003.

[10] M. C. Huebscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. ACM Comput. Surv., 40(3):1–28, 2008.

[11] Yu and Lalanda, "An approach for dynamically building and managing service-based applications architectures", IEEE APSCC, 6-10 december, 2010, Hangzhou, China

[12] Bardin, J.,Lalanda, P., Escoffier, C. 2010. Towards an Automatic Integration of Heterogeneous Services and Devices. *Proc. of* APSCC *2010.* IEEE Computer Society, Washington, DC, USA, 171-178

[13] Satyanarayanan,M. 2001. Pervasive computing: vision and challenges. *IEEE Personal Communications*, Vol. 8 (August 2001), 10-17

[14] France, R., Rumpe, B. 2007. Model-driven Development of Complex Software: A Research Roadmap. *Proc. of FOSE '07*. IEEE Computer Society, Washington, DC, USA, 37-54

[15] Levine, William S., ed (1996). The Control Handbook. New York: CRC Press