Livrable Selware SP1

Lot 1, August 14, 2007, 15:49

Selfware Platform Architecture

Contents

1 Introduction (INRIA)

Today's computing environments are becoming increasingly sophisticated. They involve numerous complex software that cooperate in potentially large scale distributed environments. These software are developed with very heterogeneous programming models and their configuration facilities are generally proprietary. Therefore, the management [1] of these software (installation, configuration, tuning, repair ...) is a much complex task which consumes a lot of resources:

- human resources as administrators have to react to events (such as failures) and have to reconfigure (repair) complex applications,
- hardware resources which are often reserved (and overbooked) to anticipate load peaks or failures.

A very promising approach to the above issue is to implement administration as an autonomic software. Such a software can be used to deploy and configure applications in a distributed environment. It can also monitor the environment and react to events such as failures or overloads and reconfigure applications accordingly and autonomously.

The goal of autonomic computing is to automate, at least in part, the functions related to systems administration. This effort is motivated by the increasing size and complexity of the systems and applications, which has two consequences: (i) the costs related to administration are taking a major part of the total information processing budgets; and (ii) the difficulty of the administration tasks tends to approach the limits of the administrators' skills.

Autonomic computing aims at providing systems with self-management capabilities, including self-configuration (automatic configuration of parameters), self-optimization (continuous performance monitoring and reaction to reach optimality), self-healing (detecting defects and failures, and taking corrective actions), and self-protection (taking preventive measures and defending against malicious attacks). Currently, human administrators perform management actions to ensure the desired operation of the system, using appropriate tools. One approach to autonomic computing, which we can call the *architecture-based approach* to autonomic computing, views the functioning of an automic computing system as an evolution of this practice, along the following lines:

- The management system observes and monitors the managed system; the observation may be active (triggered by the observer) or passive (triggered by the observed element).
- On the base of the observation results, the management system takes appropriate steps to ensure that the preset goals are met. This may entail both preventive and defensive actions, and may necessitate some degree of planning.

In a more detailed view, an autonomic computing system is organized according to the above overall scheme, sketched on Figure

[1] we also use the term administration to refer to management operations

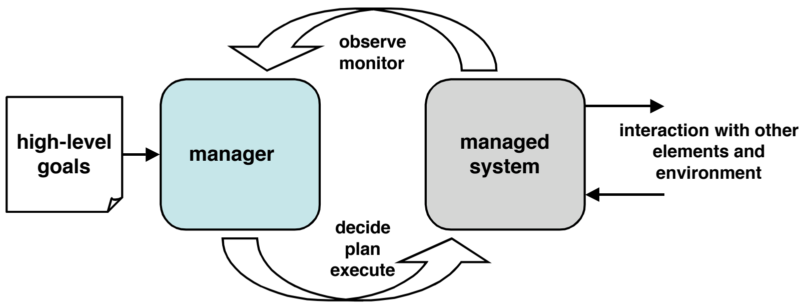

Figure 1: Overall view of an autonomic element

In this approach, autonomic computing is closely related to control. More precisely, an autonomic system is governed a feedback control loop (feedback is preferred to feedforward because it allows to take into account disturbances in the expected behavior of the controlled system).

The main advantages of this approach are:

- Providing a high-level support for deploying and configuring applications reduces errors and administrator's efforts.
- Autonomic management allows the required reconfigurations to be performed without human intervention, thus saving administrator's time.
- Autonomic management is a means to save hardware resources as resources can be allocated only when required (dynamically upon failure or load peak) instead of pre-allocated.

This document presents the Selfware platform architecture, an environment for developing autonomic management software. Selfware mainly relies on the following features:

- A component model. Selfware models the managed environment as a component-based software architecture which provides means to configure and reconfigure the environment.
- Control loops which link probes to reconfiguration services and implement autonomic behaviors.

We use the Selfware environment to design autonomic capability in the context of a clustered J2EE application [28] and in the conext of a distributed JMS infrastructure [3]. These capabilities are related to self-optimization and self-healing properties. The rest of the document is organized as follows. Section 2 presents the design principles underlying Selfware. Section 3 depicts the architecture of the management system. The design principles and the architecture is illustrated by some examples given in Section 5. Finally, Section 6 gives an overview of the Seflware common tools.

2 Design Principles (INRIA)

2.1 Basic definition

An autonomic system[29] implements a control loop that regulates a part of the system, which we call the *managed system*. The managed system is in turn constituted by a collection of *managed elements*. A managed element (ME) may consist of a single elementary hardware or software component, or may be a complex system in itself, such as a cluster of machines, or a middleware system. In order to be included in a control loop, a managed element must provide management interfaces, which include sensor interfaces and actuator interfaces[2]. These are used by a controller, also called an *autonomic manager* (AM) to regulate the managed system through a feedback control loop. The *autonomic system* (AS) is the ensemble including the managed elements and the control loop, i.e. the controller and the communication system that it uses to access the management interfaces.

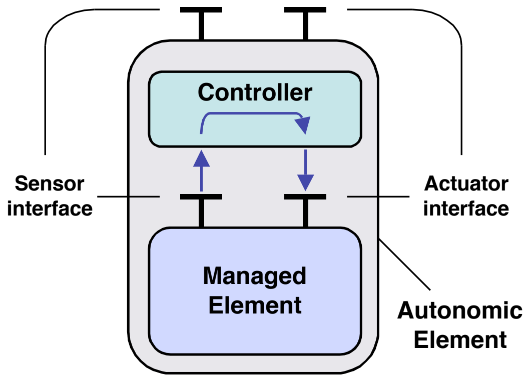

Figure 2: An autonomic element

An autonomic manager may itself be equipped with management interfaces, thus becoming in effect a managed element. This allows a hierarchical organization of AMs. In the same vein, an elementary managed element, such a hardware device like a disk unit, may itself include embedded, built-in control loops, making it an autonomic system, even if these control loops are not directly accessible through the ME's management interfaces.

The controller that regulates a ME in an autonomic element usually deals with a single control aspect, e.g. security, fault tolerance or performance. A given ME may then be part of several ASs, each of which deals with a specific aspect; each of these ASs has a specific AM and may be regarded as a different management domain.

The AMs managing different aspects of a common element have different criteria and may take conflicting decisions. An approach to resolving such conflicts is to coordinate the AMs that manage different aspects through a new AM (the coordinator), which applies a conflict management policy. An example of such a policy might be as follows, to arbitrate between a repair manager and a performance optimizing manager: give priority to the repair manager, except when this would degrade the quality of service of a specified client.

[2]The complexity of these interfaces depends on that of the managed element.

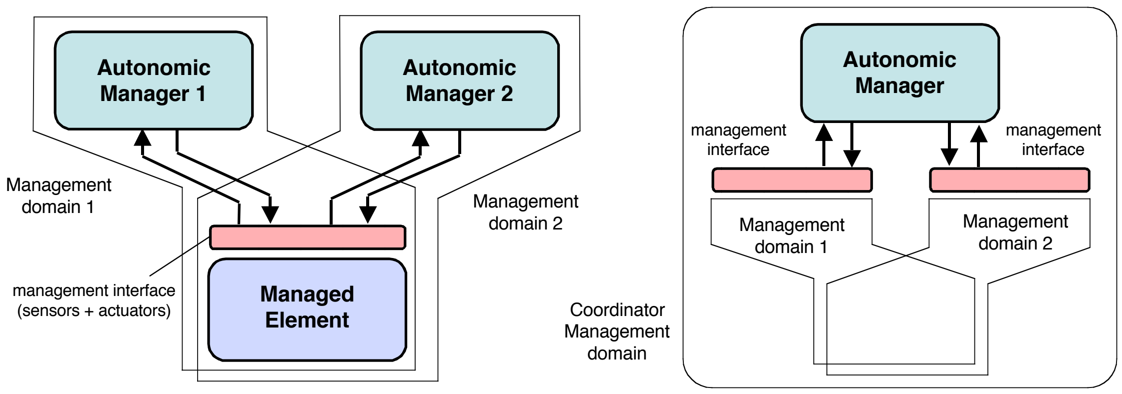

Figure 3: Multiple domains for autonomic management

The notion of an autonomic system is thus seen to cover a wide range of situations, at different levels of granularity. In the above discussion, we came across three typical levels, in increasing order of abstraction:

- An elementary component, with embedded internal control.
- A mid-level manager, at the application or midleware level, controlling an aspect such as QoS, security, or fault tolerance.
- A coordination manager, whose role is to arbitrate between several aspect-specific managers in charge of a common set of resources (a shared ME).

A managed element, be it elementary or complex, needs to provide management interfaces, including sensors and actuators. There is currently no standard defining these management interfaces, and the existing experimental platforms have developed their own set of tools (see e.g. [10]). When dealing with legacy software, not initially intended to be externally managed, a common solution for providing a management interface is to use a *wrapper*, or *adapter*. The function of the wrapper is to make a bi-directional translation between the interface provided by the legacy components and the specified management interface. This may be a best effort attempt, depending on the available interfaces of the legacy software.

2.2 Architecture-Based Management

The notion of a system model is taking an increasing importance in the design, the development and the management of software systems. A *system model* is a formal or semi-formal description of a system's organization and operation, which serves as a base for understanding the system and predicting its behavior, as well as for its design and implementation. The goal of current models of systems architecture is to describe a complex system as an assembly of elementary parts, using the notions of components, connectors, and configurations. These entities have different concrete representations according to the specific architectural model being used, but share common properties (see e.g. [23]).

As system architecture is pervading the area of systems design, it has been realized that its constructs also form an adequate base for systems management. In particular, components may be conveniently used as units of deployment, of fault diagnosis, of fault isolation, as well as domains of trust; reconfiguration is adequately represented by component replacement and connector rebinding. The notion of *architecture-based management* captures this trend. It promotes the use of architectural models and formal or semi-formal system descriptions as guidelines for various management functions. Such descriptions are becoming commonly available, e.g. in the form of an Architecture Description Language (ADL), a framework for a formal description of a system conforming to an architectural system model.

Our approach to an architecture-based, control approach to the design and construction of autonomic systems, relies first on the choice of an appropriate *component model.* The component model we use is the fractal model [13]. The Fractal component model is a reflective component model intended for the construction of dynamically configurable and monitorable systems. Its main features include: composite components (to obtain a uniform view of applications at various levels of abstractions), binding components (to reify arbitrary connections and communication semantics between components), introspection and reconfiguration capabilities (to monitor, control and modify the execution of a running system).

The choice of fractal is motivated by several considerations:

- fractal supports a hierarchical modelling of systems, allowing to describe a system at different levels of abstractions.
- fractal supports software architecture descriptions with component sharing (where components may belong to different component hierarchies), which allows more natural specifications of system structures, and a separation of concerns in architecture descriptions.
- fractal supports reflective components, i.e. components equipped with a meta-level structure that allows to monitor and control the execution of a component. The interfaces that make up a reflective component meta-object protocol in effect provide management interfaces for the component.
- fractal is an open model, that makes no predefined choice concerning the semantics of component composition. This allows a system designer to define the semantics of a component binding or of a component meta-level best suited to its design.
- fractal comes equipped with an extensible architecture description language (ADL) that can be used as a pivot language for capturing information related to different management concerns, and for expressing management actions.

We briefly present in the rest of this section the main features of the fractal component model.

2.3 The fractal component model

The fractal component model is a general component model which is intended to implement, deploy, monitor, and dynamically configure complex software systems, including operating systems and middleware. This motivates the main features of the model: composite components (to have a uniform view of applications at various levels of abstraction), introspection capabilities (to monitor and control the execution of a running system), and reconfiguration capabilities (to deploy and dynamically configure a system). A fractal component is a run-time entity that is encapsulated, and that has a distinct identity. A component has one or more interfaces. An interface is an access point to a component that supports a finite set of methods. Interfaces can be of two kinds: server interfaces, which correspond to access points accepting incoming method calls, and client interfaces, which correspond to access points supporting outgoing method calls.

Communication between fractal components is only possible if their interfaces are bound. fractalsupports both primitive bindings and composite bindings. A primitive binding is a binding between one client interface and one server interface in the same address space. A composite binding is a fractalcomponent that embodies a communication path between an arbitrary number of component interfaces. These bindings are built out of a set of primitive bindings and binding components (stubs, skeletons, adapters, etc.). The fractal model thus provides two mechanisms to define the architecture of an application: bindings between component interfaces, and encapsulation of components in a composite.

The above features (hierarchical components, explicit bindings between components, strict separation between component interfaces and component implementation) are relatively classical. The originality of the fractal model lies in its open reflective features. In order to allow for well scoped dynamic reconfiguration, a fractal component can be endowed with controllers allowing its introspection and the control of its behavior.

At the lowest level of control, a fractal component is a black box that does not provide any introspection capability. Such components, called base components, are similar to plain objects in an object-oriented language. Their explicit inclusion in the fractal model facilitates the integration of legacy software. At the next level of control, a fractal component provides a `Component` interface, similar to the `IUnknown` interface in the COM model, that allows one to discover all its (client and server) interfaces. At upper levels of control, a fractal component contains several controllers that expose (part of) its internal structure. A controller can superpose a control behavior to the behavior of the component's sub-components, including suspending, checkpointing and resuming activities of these sub-components. A controller can also play the role of an interceptor, used to export the interface of a sub-component as an interface of the parent component, and to intercept the oncoming and outgoing method calls of an exported interface. The fractal model allows for arbitrary (including user defined) classes of controller and interceptor objects. It specifies, however, several useful forms of controllers, which can be combined and extended to yield components with different reflective features, including the following:

Attribute controller: this controller allows getting and setting the component's attributes, i.e. its configurable properties.

Binding controller: this controller allows binding and unbinding the component's client interfaces to server interfaces.

Content controller: this controller supports an interface to list, add and remove sub-components in the content of the component.

Life-cycle controller: this controller allows an explicit control over the component's execution. Its interface includes methods to start/stop the component's execution.

Several implementations of the Fractal model have been issued in different contexts, e.g. an implementation devoted to the configuration of operating systems on a bare hardware (Think) or an implementation on top of the Jave virtual machine (Julia) targeted to the configuration of middleware or applications.

3 Architecture Overview

A Selfware system is organized as a collection of fractal components of three different kinds: *managed elements*, *autonomic managers* and *common services*. The extent of a Selfware administration domain is given by a set of managed elements representing physical nodes (i.e. computers running Selfware components), called *managed nodes*. In the rest of this report, we use the term *node* for managed node, unless explicitly specified (e.g. physical node). To enable self-management functions, autonomic managers and components that provide common services are managed elements in the Selfware architecture. To enable a self-healing behavior, the Selfware architecture distinguishes between two kinds of nodes: manager nodes, which host autonomic manager components, and other standard managed nodes. Manager nodes, together with their subcomponents, can be replicated to make the set of manager nodes fault-tolerant and to allow fault management policies implemented by the autonomic fault manager to apply to manager nodes as well as to standard nodes. In the rest of this section, we present a brief overview of the different kinds of Selfware components, the main functions provided by the common services, and the replication structure for manager nodes. A more detailed description of the various elements of this architecture is given in the following sections.

3.1 Managed elements

A managed element (ME) is a fractal component that encapsulates a controlled entity. It can provide sensors to collect information about the entity and actuators to change the state of the entity. We make little assumption about the controlled entity, apart from the fact that it can be manifested as a fractal component. A managed element can be a software or a hardware resource; a primitive or a composite component; a component located in a single address space, or a distributed composite[3]. Managed nodes are managed elements that encapsulate physical nodes. A managed node provides interfaces to install component packages and to create managed elements on the node.

3.2 Autonomic Managers

An autonomic manager (AM) is a fractal component that implements the analysis, decision, planning and execution stage of a control loop: it monitors a set of managed elements, analyzes notifications coming from managed element sensors, diagnoses the state of the system, decides on or plans a course of action in response to the diagnosis, and according to high-level administration policies it obeys, and executes the corresponding command

[3]In the current version of the Selfware system, managed elements correspond to non distributed components. A managed element is either a managed node, or a component that executes on a managed node.

plan. The managed elements controlled by an autonomic manager may be designed implicitly or explicitly. Since an autonomic manager is also a managed element, it can be remotely deployed/undeployed and it can be controlled by another autonomic manager.

An autonomic manager can rely for its operation on a system view maintained by the system representation service (described below). Two principal modes of operation are possible with respect to the system representation. In the passive mode, the system representation is updated by the autonomic manager according to notifications received from managed elements sensors, and commands that it executes on managed elements actuators. In the active mode, notifications and commands are mediated by components in the system representation that implements proxies to managed elements.

3.3 Common services

Common services in the Selfware architecture provide a set of basic capabilities used by autonomic managers to perform elementary configuration management functions, such as installing and deploying components on nodes, or communicating with sensors and actuators. Components that provide common services are also managed elements. The common services described in this report are the following :

- The resource allocation service provides the functionality to allocate resources for the managed system as well as for the management system. In a first step this service will be used for node allocation. Node allocation can take several forms, and can provide more or less sophisticated levels of hardware resource virtualization. The node allocator service is typically used by autonomic managers to request or relinquish nodes.

- The navigation service is designed specifically to express queries on Fractal architectures. These queries can be used to "walk" inside a running Fractal application, discovering its structure and selecting elements according to any property represented in the Fractal model. This service allows an easy access to the architecture of the managed system as well as of the management system. It complements the system representation presented below.

- The system representation service is used to maintain a consistent view of the runtime system as a component-based architecture. This view is isomorphic, introspectable and causally connected to the runtime infrastructure.

- The deployment service aims at deploying both the managed system and the management systems on remote nodes. This service (i) download software packages from a remote location. (ii) install the required software on the cluster, (iii) configures, instantiates and starts managed elements as well as removes managed elements from the infrastructure.

- The monitoring service is used to gather arbitrary level information about resource execution and aggregate these informations to provide high level event with more semantics. The monitoring service is composed of probes which represent the sensor of the infrastructure.

- The decision service provides decision making tools (models, languages and runtime) for implementing the reactive part of autonomic management policies defined by autonomics managers in their control loop.
- The wrapping service provides the language and the runtime used to generate the component wrappers used to control legacy software. This service will generate fractal component from wrapper description.
- The reconfiguration service provides two separate but related needs for the Selfware platform with respect to reconfiguration. First, a specific language (DSL) to ease the description of reconfigurations. Second, we need to be able to define correctness for these reconfigurations, which might depend on the target application, and to guarantee that only correct reconfigurations can be applied, as efficiently as possible.

Figure 4 gives an overview of the Selfware architecture.

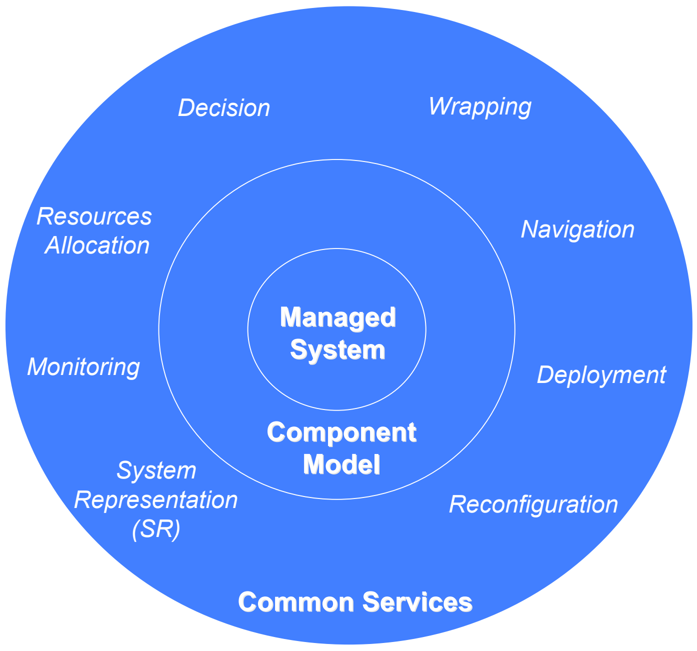

Figure 4: Selfware architecture overview

3.4 Replication structure for manager nodes

The replication of manager nodes is in response to the need for fault tolerance of autonomic managers, and to the need of a self-healing system. Realizing a self healing system in the control approach we have adopted for Selfware requires to establish a meta-level control loop for monitoring and dealing with manager node failures. We build this meta-level loop by replicating manager nodes in a Selfware system, which provides also for fault tolerance.

The replication structure, illustrated in Figure 5, is obtained by: (1) actively duplicating the Manager component on different nodes, and (2) by ensuring that the System Representation contains a representation of the Manager nodes themselves (together with their internal configuration and bindings, as for other nodes). A standard active replication of the Manager component on different nodes is not sufficient, however, for one must avoid the duplication of commands.

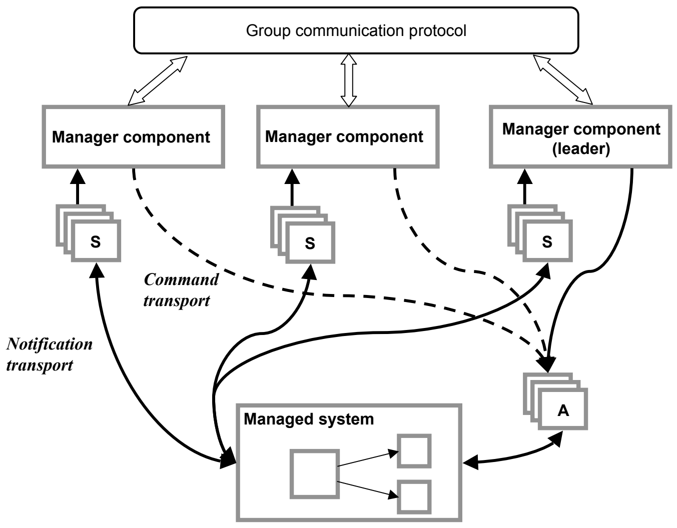

Figure 5: Replication structure

The active replication of the manager nodes can be realized according to the following scheme, using a uniform atomic broadcast protocol with a leader role:

- Manager nodes (together with their autonomic managers, system map and node fault detectors), are replicated according to the required level of fault-tolerance (if we only consider silent failures with no possible network partition, f manager node failures can be tolerated using $f + 1$ nodes).

- We assume the node fault detection service is available on all manager nodes, and that it can provide the guarantee that a node crash is ultimately detected by at least one non failed manager nodes.

- Each node failure notification to, and each command from, a manager node, is broadcast to all other manager nodes in the system (manager nodes form a single broadcast group in the system), using the uniform atomic broadcast protocol.

- Only the autonomic managers of the Leader manager node act on failure notifications or instructions from the Console. In particular, only the Leader autonomic manager sends requests to the node allocator and issue configuration commands to managed nodes.
- Associations between notifications and resulting commands, as well as between console instructions and resulting commands, are assigned unique identifiers and journalized by autonomic managers in each manager node. This allows, for example, the repair manager of a newly elected Leader to complete the reaction to a failure notification which has not been completed by the failed former Leader.
- Identifiers for notification/command pairs uniquely identify commands, and are sent as additional parameters of each command. Actuators discard commands they receive that bear the same identifier of a previously executed command. This takes care of the potential window of vulnerability that exists between the completion of a command and the failure of a Leader manager node.

4 Managed element

This section introduces two kinds of managed elements : (i) the legacy managed element and (ii) the architecture managed element.

4.1 Legacy Managed Element

A Legacy Managed Element (LME) is any piece of legacy system; it can be a hardware element such as a cluster node, or software element such as a middleware running on a node or an application running on the middleware. A Managed Element is equipped with a uniform management interface that provides a homogeneous view of all the managed legacy elements. This allows Autonomic Managers to apply common self-management policies to different managed elements (e.g. different legacy systems). The basic uniform management interface is composed by the fractal interfaces described above : (i) Attribute controller interface (property management and introspection). (ii) Binding controller interface (Interconnection management and introspection). (iii) Life-cycle controller interface (Life cycle management and introspection).

When dealing with legacy software, not initially intended to be externally managed, a common solution for providing such management interface is to use a *wrapper*, or *adapter*. The function of the wrapper is to make a bi-directional translation between the interface provided by the legacy components and the specified management interface. Thus any software managed with Selfware is wrapped in a Fractal component which interfaces its administration procedures, through the provision of controllers. This provides a means to manage legacy entities using a uniform model (the Fractal control interface), instead of relying on software-specific, hand-managed, configuration files. Furthermore, it permits to add a control behavior to the encapsulated legacy entities (e.g. monitoring, interception and reconfiguration). All wrappers components provide the same (uniform) management interface for the encapsulated software, whereas the corresponding implementation is specific to each software (e.g. in the case of J2EE, Apache web server, Tomcat Servlet server, MySQL database server, etc.).

Wrappers are the part of the management system specific to the legacy software while they provide the same basic fractal API. Relying on this management layer, sophisticated administration programs can be implemented, without having to deal with complex, proprietary configuration interfaces, which are hidden in the wrappers.

4.2 Architectural Managed Element

An Architectural Managed Element (AME) represents a given architectural organization of Managed Elements, e.g. hardware/software stacks (computer, OS, middleware, application), or the client/server architecture. Thus, Architectural Managed Elements are the general representation of complex system architectures. Thanks to Fractal's hierarchical model, arbitrary sophisticated organizations can be modelized and managed using appropriate composite components. Relying on the fractal composition mechanism, standard fractal controllers provide coarse grain management operations over the composite structure. For instance network topology, cluster of machines, distributed J2EE infrastructure can correspond to a composite component at runtime. These coarse grain management operations are independant from any applicative context :

- Architecture model and introspection : The software architecture is present at runtime as a set of interconnected component. This architecture provides a basic knowledge which can be used by control loops through introspection operation. Introspection allows to discover the structure of a complex infrastructure materialized at runtime by a composite component. We can discover the subcomponents, their interconnection, their configuration, their lifecycle. Note that the architecture remains consistent when the legacy system is updated.

- Architecture deployment : Deploying a composite component induces the, possibly remote, deployment of its subcomponent and interconnection. This level of deployment is independant from the legacy since configuration specificity are tackle by the wrappers (i.e : the basic managed element).

- Architecture lifecycle : These operations allow to manage the lifecycle of the complex structure corresponding to the composite inner architecture. The standard implementation of the lifecycle controller implements a generic workflow (depth first search) to start and stop subcomponents. This implementation can be modified to ensure specific workflow if necessary.

- Architecture reconfiguration : It provide operations to reconfigure the composite inner architecture. It can be used for instance to add or remove subcomponent in a composite structure.

5 Example

5.1 Legacy Managed Element

We now give an example of a Fractal wrapper for the Apache server that is part of the J2EE architecture described above. As a managed element the wrapper provides an attribute controller, a binding controller and a lifecycle controller:

- *The attribute controller interface* is used to set attributes related to the local execution of the Apache server. For instance, a modification of the port attribute of the Apache component is reflected in the *httpd.conf* file in which the port attribute is defined. This interface can also be used to retrieve some specific configuration properties.
- *The binding controller interface* is used to connect Apache with other middleware tiers. For instance, invoking the bind operation on the Apache component sets up a binding between one instance of Apache and one instance of Tomcat. The implementation of this bind method is reflected at the legacy layer in the *worker.properties* file used to configure the connections between Apache and Tomcat servers. This interface can also be used to change a connection between two component and to discover existing bindings.
- *The life cycle controller interface* is used to start or to stop the server as well as to read its state (i.e. running or stopped). It is implemented by calling the Apache commands for starting/stopping a server.

Other servers (Tomcat and MySQL) are wrapped in a similar way into Fractal components, and provide the same management interface.

Once wrapped, many instance of legacy software can be combined together to form a complex distributed infrastructure. The component layer provide : (i) a naming system to identify software instance, (ii) a type system to ensure the consistency of software interconnection. In the following, we show the benefits of using Selfware to perform system reconfiguration, compared to an ad-hoc approach. Figure 6 illustrates a scenario where, initially, an Apache web server (Apache1) is running on node1, and connected to a Tomcat Servlet server (Tomcat 1) running on node2. In this scenario we want to reconfigure the clustered middleware layer by replacing the connection between Apache1 and Tomcat1 by a connection between Apache1 and a new server Tomcat2.

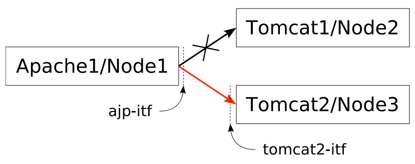

Figure 6: Reconfiguration scenario

Without the Selfware infrastructure, this simple reconfiguration scenario requires the following steps to be manually done (with very low reactivity), in a legacy-dependent way:

first log on node1, then stop the Apache server by running the Apache *shutdown* script, then edit and update the configuration file (*worker.properties*) in Apache to specify its binding to the new Tomcat server (Tomcat2 on node3) as follows:

```
worker.worker.port=8098
worker.worker.host=node3
worker.worker.type=ajp13
worker.worker.lbfactor=100
worker.list=worker, loadbalancer
worker.loadbalancer.type=lb
worker.loadbalancer.balanced_workers=worker
```

Finally, the Apache server is restarted by running the *httpd* script. With Selfware, as soon as the required wrappers have been implemented, such a reconfiguration can easily be implemented in an administration application. The operations required to perform this same reconfiguration are simply few operations on the involved components in the management layer, namely:

```
Apache1.stop()
  // unbind Apache1 from Tomcat1
Apache1.unbind("ajp-itf")
  // bind Apache1 to Tomcat2
Apache1.bind("ajp-itf",tomcat2-itf)
  // restart Apache1
Apache1.start()
```

In the following we give the example of an apache wrapper (we focus on the lifecycle controller).

```
// Here is the standard interface for the lifecycle controller
public Interface LifeCycleController {
 /** Starts the component to which this interface belongs */
 public void startFc();

 /** Stops the component to which this interface belongs */
 public void stopFc();

}

// Here is the wrapper code that implements this interface for the apache server

public class ApacheWrapperImpl implements LifeCycleController, ... {
...
...
  public void startFc() throws SelfwareException {
    ShellCommand.syncExec(dirInstall + "/bin/httpd -f" + dirLocal + "/conf/httpd.conf");
  }

  public void stopFc() throws SelfwareException {
    BufferedReader br = new BufferedReader(new FileReader(new FileReader(dirLocal + "/log
    ShellCommand.asyncExec("kill -TERM" + br.readLine());
  }
}
```

5.2 Architectural Managed Element

In the case of a J2EE infrastructure, we can provide a J2EE component built from a set of sub-components representing the legacy servers (apache, tomcat, mysql) and their interconnections. This is illustrated in Figure 7. In this setting,the structure of the J2EE component is composed by : (i) a L4-switch balances the requests between two Apache server replicas, (ii)Two Apache servers connected to two Tomcat server replicas, (iii) Two Tomcat servers both connected to the same MySQL server. The vertical dashed arrows (between the management and legacy layers) represent management relationships between components and the wrapped software entities. In the legacy layer, the dashed lines represent relationships (or bindings) between legacy entities, whose implementations are proprietary. These bindings are represented in the management layer by (Fractal) component bindings (full lines in the figure).

In the case of this J2EE component , an administration program can inspect the overall J2EE infrastructure, considered as a single composite component (i.e. to discover two Apache servers interconnected with two Tomcat servers connected to the same MySQL database server). It can also add or remove legacy servers in the J2EE infrastructure (for instance adding a tomcat server between an existing set of apache servers and databases). We can also invoke the start/stop operation provided by the lifecycle controller of the J2EE component. This will start/stop all the middleware servers belonging to this component (the apache servers, the tomcat servers and the databases) according to the implemented lifecycle workflow.

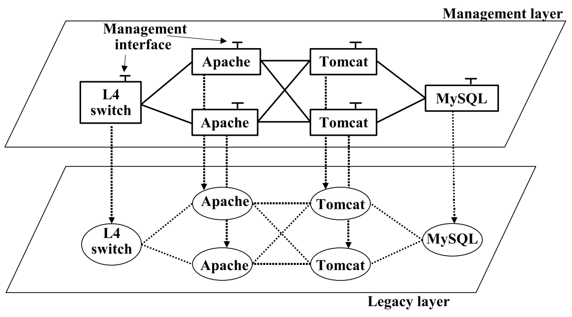

Figure 7: Management layer for a clustered J2EE application

6 Common Services (ALL)

6.1 Deployment (INRIA)

6.1.1 Goal

The goal of the deployment service is to install and configure application and well as to undeploy them. The deployment service will deploy application's components according to (i) the ressources provided by members and (ii) some contrainsts related to the application (for instance locality constraints...). in this section we provide a functionnal description of the deployment service.

The deployment service is architecture-based. It means that given a description of the software architecture, the deployment service is capable to install, instantiate and run software components described by this architecture. Moreover, the deployment system allows for component versioning and dynamic updates – several versions of software components can coexist, components can also be replaced with their new versions.

The software elements to be deployed must be described using the Fractal ADL. The next section present the requirement of the deployment service. All these requirement are not currently fulfilled by using a standard ADL factory.

6.1.2 Requirements

- The user can insert or remove a component into an existing component structure.
- The configuration of a component can be explicit or implicit.
- The deployment service must be able to deploy complex component structure which can be made of legacy components.
- Non functional properties can be easily plugged in the deployment process. For instance an interesting property is to ensure the atomicity of deployment orders.
- The deployment service must be fully specializable. In particular, the implementation and the scheduling of the deployment orders can be controlled by the user.

6.1.3 Deployment workflow

This service is implemented using an ADL Factory. The deployment algorithm requires that physical nodes are wrapped as managed elements and provide (i) an installation API to install software on the node and (ii) a component factory API to create managed elements. The algorithms of the `deployment` process is summarized below:

- Lookup for the node where the component must be deployed.
- Install the component package on the target node.
- Instantiate the component on the target node.
- Configure the component (i.e its attributes and its external bindings).
- Process recursively with the sub-components if necessary.

The basic deployment steps are depicted on figure

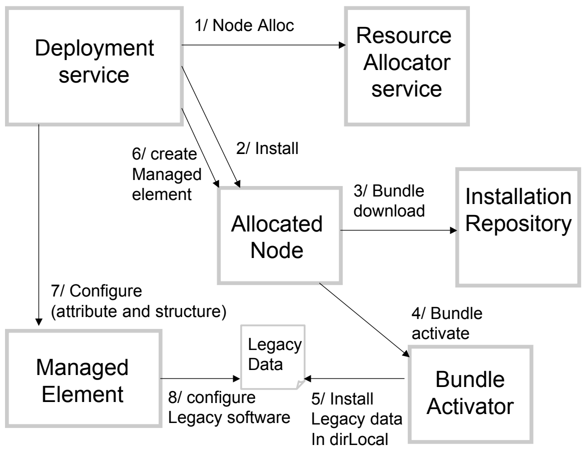

Figure 8: The steps of the deployment process

6.2 Wrapping (IRIT)

Wrapping is one of the common services of the Selfware architecture. Wrappers can be used to encapsulate diverse software resources so that they all present a common and simplified interface. Software wrapping is a technique in which an interface is created around an existing piece of software, providing a new view of the software to external systems, objects, or users. Using wrapping techniques, legacy software (e.g. Apache, Tomcat, MySQL etc) can be changed to software components that can be integrated (to new computing environment), manipulated (installed, configured, re-configured), controlled and maintained (repaired) automatically and in efficient manner. In the context of Selfware, wrapping helps developing an environment whereby a potentially large number of existing legacy software components can be adapted for use within a new software administration framework. Wrapping facilitates self-managing capabilities such as self-configuration, self-optimization, self-healing, and self-protection which are very important properties of autonomic computing.

6.2.1 Basic principles of wrapping

The aim of wrapping, in the context of Selfware, is to encapsulate the different components of the J2EE and JMS legacy software platforms in order to develop administration systems with capabilities of performing the following activities in automatic, efficient and dynamic fashion:

- Installation and deployment
- Configuration (re-configuration)

- Error detection and maintenance

The basic idea of wrapping is to provide control mechanism to do the above activities by the system. The internal activities of the legacy systems or the interactions/calls between them are not intercepted or controlled by the system. The system facilitates only the configuration of the different components.

6.2.2 Fractal model and wrappers

Since the Selfware system is built on the Fractal component model, the purpose of the wrappers is to create a corresponding Fractal component for each component of the legacy software. The Fractal model provides adequate support to develop wrappers for legacy software components because it provides well defined interfaces such as attribute controller, binding controller, content controller and lifecycle controller which can easily be mapped to the installation, deployment, configuration and maintenance activities in the legacy software. Once one legacy software component is wrapped into a Fractal component, the manipulation of the legacy software is done through the different control interfaces of the Fractal component.

Consider a clustered J2EE architecture composed of Apache, Tomcat and MySQL components. In such a scenario, normally, a human administrator configures the architecture by accessing the configuration files associated with each component. For example, to configure the port of the Apache server and to define the binding or connection of the Apache server with other components, this requires the modification by the administrator of the httpd.conf and the worker.properties files respectively. With the Selfware infrastructure, by wrapping each of the legacy software into Fractal components, those activities can be done in an automatic and dynamic manner by an administration system.

For example by creating a Fractal wrapper for the Apache server, the attribute controller interface is used to configure the properties of the Apache server and the binding controller interface is used to define the connection of the Apache server with other components of the system, in this case Tomcat servers. At the same time the life cycle controller interface is used to start, stop and read the state (running or stopped) of the server.

We identify the following related works on wrapper development, which is a potential source of inspiration.

- *Kilim* (http://kilim.objectweb.org/). Kilim is a configuration framework based on the Fractal component model, that provides a generic model, a powerful language and tools (a parser, a runtime configuration viewer) to facilitate, automatize and control the configuration process of arbitrary complex applications. Kilim enables the definition of composable abstractions called templates, capturing encapsulated sub-systems, defining their properties and their connectivity (slots), defining the mapping of these abstractions to existing code in terms of constructors used to create the various instances they may contain and setters/methods used to configure and to connect them; and the recursive assembly of these components allowing to build complex systems. Here, to build a complex system a template is created by assembling existing templates. Kilim creates components from Java code using the templates.

- *Cargo* (http://cargo.codehaus.org/Home). Cargo is a thin wrapper around existing containers (e.g. J2EE containers). It provides different Java APIs to easily manipulate containers: starting/stopping containers, configuring containers for deployment on any user-specified directory, deploying WAR and EAR components on these containers.

6.2.3 Requirements

Wrapping is one of the common services of the Selfware architecture. We aim at providing a wrapping service with two components: a wrapper description language (WDL) for the specification of the wrapper associated with a legacy component, and a wrapper generator (WG) which generates the wrapping software from the WDL description. The specification provides vital information that is required to generate the wrapper of the legacy software. The wrapper generators creates the appropriate Fractal component that corresponds to the legacy software based on the specification/description.

To aid the development of an appropriate scheme for the wrapping activity, the following requirements should taken into account.

- Dedicated language. To facilitate the definition of wrappers, a dedicated language is required to specify the translation of administration interfaces into legacy software administration functions (which are proprietary). The administration interfaces of the wrappers are used by the system to control and configure the legacy software.
- Extensibility. Since the wrapped legacy software are very heterogeneous (very different administration functions), the WDL should be extensible in order to accommodate to very different classes of legacy software (and classes of administration functions). However, we believe the few personalities of WDL should cover most of the application domains.
- Generate Fractal components. The Selfware platform is based on the Fractal component model. From a WDL specification, we must automatically generate the Fractal component that corresponds to the wrapped legacy software.

To conclude, if the wrapper specification language is more generic, it has a wide applicability to very different application domains, but required more efforts from the developer to program the wrapper. If the wrapper specification language is more specific, it limits its applicability to different application domains but requires less programming efforts. The good trade-off is probably to provide a wrapping framework for defining specialized WDL languages (and their associated generators) for different applications domains.

6.3 Monitoring requirements

6.3.1 Objectives

This chapter aims at defining the monitoring requirements for the Selfware platform in terms of system observation and information feedback. These requirements come from the project's autonomic scenarios in the fields of self repair, self configuration and self optimization. Roughly, the Selfware autonomic infrastructure needs to know what the system it is managing is made of, and what is the state of this system. This information quest may follow a variety of modalities according to the target resources (software element, hardware element, network, execution support abstractions...) and the target features (deployment, configuration, repair, or optimization).

Beyond this information extraction and delivery role, the software architecture of this monitoring functionality must be consistent with and integrated to the overall Selfware infrastructure. One of the key ideas behind this requirement is that it should also benefit from the autonomic management features of the Selfware platform.

6.3.2 Required features

Generalities : The monitoring feature aims at observing the managed system and delivering the observation results to the autonomic control features, either by broadcasting events (push mode), or by keeping the information that will be picked by the interested elements when needed (pull mode). For instance, an autonomic control element may be interested in receiving an event when a system load threshold is reached, while periodically checking the availability of a computer on the network.

Elements of the monitoring functionality, that we will call probes, are likely to be active, possibly to get information (observation, measurement), and possibly to send the extracted information. Probes may also require a memory feature to keep the extracted information during a certain amount of time, either as raw data or as statistical values on sliding time frames.

Basic observations : Scenarios about self repair basically need fault detectors, such as heart beat (I'm alive!) or ping (are you alive?). Self optimization scenarios needs are:

- system load measurements (CPU consumption, RAM utilization, disk and network transfer rates, etc.),
- observation of middleware and applications involved in scenarios based on Jonas and Joram (Java Virtual Machines, Servlet servers, databases, etc.)

Aggregated observations : Beyond those basic observations that are obtained from basic probes dedicated to specific software or hardware elements of the managed system, there is also a need for more elaborate, higher level indicators computed from the basic observations. For instance:

- a system load or an application server load metric
- an aggregation of CPU loads for a given cluster of computers

Both kinds of observation may be required at the same time, and may evolve in time. Moreover, it might be necessary, or at least convenient, to share probes among several combinations. For instance, a CPU probe may be used as is, for local CPU observation, as well as simultaneously in a local system load aggregation and a CPU load aggregation for a cluster of computers. Finally, probes and aggregations should be reconfigurable.

6.3.3 Software architecture requirements

Uniform probe representation : The architecture should feature a uniform representation of all probes. This means that all probes must expose identical interfaces, even when they were in charge of monitoring different system proprieties and/or heterogeneous resources. The monitoring framework should support various probe types for monitoring different system resources and proprieties, at different abstraction levels. For example, probes could be available for monitoring a system's CPU, a JVM's memory, an application's workload, or a cluster's general load. However, all probes should be equally accessible via identical interfaces, in order to retrieve monitoring data or apply control commands.

More generally, the monitoring functionality must be deployable and manageable similarly to any other element in the Selfware platform. As a consequence, we must rely on a uniform architecture based on Fractal components that conform to the Manageable component type.

Recursive, hierarchical probe composition : The architecture must support probe organisation into recursive, hierarchical constructs. This allows system managers to build arbitrary monitoring hierarchies based on individual probes. Probes can be basic or composite. Basic probes extract actual data from the monitored system elements and represent leaf nodes in the monitoring hierarchy. Composite probes collect data from lower-level probes, which can be in turn basic or composite. Collected data is being processed so as to provide a higher-level monitoring view of the corresponding resources.

Extensibility : The architecture should enable the monitoring framework to be seamlessly extended, in terms of probe types, data formats, processing capabilities and communication protocols.

- Probe types : it should be possible to incrementally provide additional probe types for monitoring new relevant resources and system properties. The architecture should allow such additional probes to be seamlessly integrated with the existing monitoring infrastructure. Third-party or legacy probes should also be supported, provided they implemented the required framework interfaces. Specific wrappers or adapters should be specified and built in order to allow various third-party or legacy probes to be represented as Selfware probes and subsequently reused in the framework.

- Data formats : Probes should be able to provide monitoring data at various detail levels and in various formats. For example, a simple long[] array can be used to repeatedly retrieve the most frequently changing monitored parameters. In addition, more extensive monitoring data can be available in XML format, to provide a comprehensive view of all monitored parameters. In this scenario, clients can subsequently decide which data format to retrieve at various times, so as to meet their requirements while optimising performance

- Data processing : The internal probe architecture must allow data-processing functions to be extensible and configurable across different probes. This implies support for new aggregation, filtering and scheduling strategies, or even of new ways of combining such strategies into more complex data-processing chains. Upon its instantiation, each probe should be configurable with a custom data-processing chain, involving specific aggregation, filtering and scheduling functions.

- Communication protocols : Communication protocols intervene at two levels in the monitoring framework. First, they are used for inter-probe communication. Various inter-probe communication protocols should be fairly easy to set, at different hierarchical levels. Second, different communication protocols can be used for accessing and/or controlling the probes. Clients should have several choices for remotely accessing and managing probes, including JMX, JMS or RMI.

Scalability : The monitoring framework should scale gracefully with the number of monitored resources and aggregated data sources. This means that the overall monitoring hierarchy should withstand increasing numbers of monitoring nodes and that each composite probe should be able to handle large numbers of data sources.

Adaptability and robustness : The monitoring framework should be able to correctly handle the dynamic addition and/or removal of monitored resources and/or subordinate probes. This implies that system managers can dynamically instantiate or remove basic probes for monitored resources, or composite probes for higher-level properties. Also, existing composite probes should be dynamically reconfigurable so as to accept additional data sources, or to adjust their data-processing functions. Finally, composite probes should be able to withstand system resource failures that imply the sudden absence of basic probes data sources.

Manageability : The monitoring architecture should enable system managers to fairly easily control monitoring probes, probe configuration and hierarchical inter-probe bindings. As the monitoring hierarchy's scale and complexity increase, the number of manual management procedures should be minimized.

Defining autonomic behaviours for the Selfware platform requires higher level programming interfaces and abstractions than the plain probe infrastructure, in order to easily implement the autonomic control loop for a variety of use cases 7. In other words, the Fractal-based probe infrastructure provides a sound, generic architectural support that we may adapt through one or several software layers supporting high level features such as:

- Multi-criteria designation of components (structural criteria, type, name, etc.)
- Event designation and access to the values they hold
- Numerical expression definition, computed from the values hold by events;
- Boolean conditions evaluated from values hold by events or according to the events source component

Portability : The monitoring framework should be portable across various application platforms, operating systems and CPU architectures. This implies that different monitoring probes in the hierarchy should be deployable and usable on different systems, with dissimilar software and hardware characteristics.

Performance : The performance overheads induced by monitoring probes on the managed systems should be minimized. A popular approach is to minimize overheads caused by instrumentation code (i.e. for basic probes) on the actual monitored nodes and to use separate stations for performing the remaining data-processing and management functions. In addition, the performance of data transmission and processing procedures should be such that the Selfware framework can effectively learn of relevant system changes and react in due time.

Component-based architecture : The monitoring service by itself is likely to become a complex, distributed infrastructure by itself that would deserve relying on the autonomic services of the platform. To achieve this, the monitoring service shall be based on components that conform to the Manageable component type. Moreover, functional requirements in terms of composition (for aggregate observations), sharing and runtime reconfiguration also advocate for an advanced component model featuring recursion, sharing and reflexion.

Common Monitoring Utilities : Some of the most commonly required monitoring utilities should be provided to assist administrators with using the monitoring framework. Such utilities can include graphical user interfaces, probe instantiation and remote deployment facilities.

6.4 Requirements for a monitoring integration middleware

Numerous monitoring solutions already exist in many domains (network and grid, performance, application-level, environment, etc.). Each of these solutions represent real expertise in a specialized domain, but they can not be easily combined to provide an overall picture, as they are generally ad hoc and heterogeneous. Building an autonomic control loop requires a systemic view of the target application and its environment, which covers domains currently targeted by different solutions. In order to build such a systemic view without re-implementing already existing solutions, one must be able to integrate different monitoring technologies (called "providers" from now on) in a uniform way. In a way, what is needed is a monitoring middleware, able to provide a uniform–and hopefully simple–interface to a set of heterogeneous monitoring solutions.

The requirements presented below supplement and/or specialize those presented above in the case of a monitoring middleware specifically designed to integrate existing but heterogeneous monitoring solutions.

The first, core requirement is the need for a common, generic and flexible data model to represent the monitoring data from all the providers into a single, uniform way to the end user. The model must be generic, i.e. not domain specific, as the very goal is to integrate providers specialized in different domains. Because all these providers have been developed independently, the common model must be flexible to accommodate their differences. For the same reasons, the model must impose as little semantic as possible, i.e. at its core it must be closer to a plain data structure than to a predefined ontology. However, that semantic should not be imposed does not mean it should be forbidden: the model may optionally support ways for providers to declare constraints on how they use the generic model, for example using typing annotations or schema-like constructs Finally, as the model will be directly exposed to the end user, it must offer a good compromise between richness/structure and simplicity.

As already mentioned in section 1, this shared data model must be accessible to the end-user in two ways:

1. a pull mode, where the user queries the middleware to know the current structure and state of the model;

2. and a push mode where the user can ask to be notified automatically and asynchronously of specific conditions occurring in the model, so that he does not have to do potentially inefficient polling of the model.

The interface presented to the end-user, in this case a programmer, should be simple to learn and to use. This means the API should have a "low surface area" (minimize the number of functions to learn), and should try to reuse/leverage concepts already familiar to the user. It also means that the API should isolate the programmer from the actual implementation details, especially since they will be different from provider to provider. This last requirement implies that the middleware should provide different interfaces to the end-user, who needs a simple and uniform view, and the configurator, who builds this common view by choosing, configuring and integrating the appropriate set of providers. The interface provided to the configurator will be more complex than the one presented to the user, but this complexity should be hidden from the latter.

The middleware should support real and seamless integration of heterogeneous providers, meaning that it should not just be a common "wrapper" API, but allow multiple providers to coexist at the same time inside a given instance of the middleware. Also, these multiple providers should not be completely isolated from each other, but be able to interact, for example when the configuration of one provider must be updated to reflect new information from another (e.g. when one provider detects a new device plugged in the system, another wants to be notified so it can deploy its own probes on the new device).

As the objects monitored by the system can be highly dynamic and change unpredictably, the middleware configuration must itself be dynamic so as to be adapted to reflect changes in the target system. This dynamic reconfiguration must be controlable by program so that it can be automated (for example as part of an autonomic control loop) and so tools can be built for configurators/administrators.

The middleware should be able to support both simple providers, i.e. primitive probes which monitor a single element and provide simple measures, as well as structured providers which have their own, potentially complex, structured data model which must be mapped into the common model.

The middleware should support abstraction on the data obtained from the providers: it should not be limited to importing raw measures given by a provider, but should also be able to adapt/abstract over these raw values (by computing derived/aggregated values) to offer a less "idiomatic" view. Examples include changing the unit of measure used by a provider or computing a derivative (bandwidth used from the total number of bytes sent/received).

Finally, the middleware should impose as little overhead (both in memory and CPU) in the use of the integration framework/middleware compared to a "native" usage of the provider.

6.5 Decision (BULL, FT)

Decision making mechanims provides the tools (models, languages and runtime) for implementing the reactive part of *autonomic management policies* defined by autonomics managers in their control loop. Each management policy is "distributed" across the different functions of the architecture:

- in the monitoring function for extracting the relevant information. A part of the filtering and aggregation task can be done in the monitoring feature.
- in the analysis function for providing the mechanisms that correlate and model complex situations (for example, time-series forecasting and queuing models). These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations. The analyse function evaluates the different conditions that aims to update the global state of the system. This state and its changes are used as inputs in the condition part of the plan's rules.
- in the plan function for providing the mechanisms that construct the actions needed to achieve goals and objectives. The plan function applies the adaptation policy and fires the rules acting on the system. Depending of the complexity of the operation, the number of steps, actions can be organized in a workflow process. In this case, the plan rules throws an action part that creates an instance of a process. The different interactions at each task can be held by others rules instead of human.
- in the execute function by providing the mechanisms that control the execution of a plan.

Two approaches are investigated in the Selfware context for decision making and more globally for reactive capabilities: one is based on *active rules* (Event Condition Action or ECA rules), the other is based on *production rules.* The initial hypothesis in the Selfware architecture is that an active rules mechanism would be used in local control loops where reactiveness in essential ; while an production rules mechanism would be used for global (centralized) management where production rules will be expressed in a high level language close a natural language. The global decision service would make use of a workflow engine that can be used for implementing the plan function.

This objective of the work on decision making in Selfware is i) to study comparatively the two approaches, ii) to validate or invalidate the above hypothesis and iii) more globally to study interactions between local and global decision making process.

6.5.1 Active (ECA) rules

Objective Reactive behaviour, the ability to act/react automatically to take corrective actions in response to the occurrence of situations of interest (events) is a key feature in autonomic computing. This reactive behaviour is typically incorporated by *active rules* (Event-Condition-Action or ECA rules) [40] a mechanism widely used in active database systems to provide a reactive behaviour (an elaborated form of *triggers* as found in most commercial DBMS). Active rules in ADBMS are used for the implementation of integrity constraints, derived data, update propagation, default values, versions and schema evalution management, authorisations, etc.

The approach followed here consists in defining a mechanism for the integration of active rules in component-based systems to augment them with autonomic properties. The fundamental idea being to "extract" the reactive functionality of active database systems[24], and to "adapt" and "inject" it for component-based systems so as to provide them with autonomic capabilities.

Rational Active rules in database systems have been extensively studied but cannot be directly applied to component-based systems. Three main points deserve a special attention in this respect:

- the definition of a active rule definition model suitable for component-based distributed systems. In active database systems, events, conditions are actions are essentially related to data manipulation through (SQL) query statements - while in component-based autonomic systems, events, conditions and actions are essentially related to interactions between components (operation invocations on component interfaces),
- the definition of a rule execution model suitable for component-based distributed systems. In active database systems, rules are triggered by events generated in the context of a transaction, conditions are evaluated and actions executed in the context of a transaction as well (the three steps in one unique transactions or in concurrent transactions). All dimensions/parameters of rule execution in active database systems are also based on the presence of transactions in ADBMS which represent a natural and convenient execution unit. Transactions are generally absent in autonomic component-based systems. Active rule execution models in ADMBS have to be re-visited for component-based systems.
- the architectural integration of rules in component-based distributed systems. In active database systems, rules are generally represented and manipulated as any other data: typically relations (tables) in relational DBMS or objects in object-oriented DBMS. Their scope is global to a database schema (a set of relations in relational DBMS, a set of persistent classes in object DBMS). In component-based systems, the nature (e.g. implicit rules implemented as part of a component platform or rules as components) and the scope (a rule attached to one component or rules with broader scopes) of rules have to be stated.

Also important, the extensive study of active rules in database systems has shown that one semantics (specify by an execution model) does not match all applicative needs. On the contrary, what is needed are flexible execution models that allow programmers to adapt the rule execution semantics to their specific needs. A overall objective is then to come up with an adaptable architecture that would support flexible rule execution models.

Reference models and architecture As this work is underway, we can draw here the big picture of the envisioned ECA rules mechanim:

Definition Model The rule definition model specifies the form (format) of events, conditions and actions. Considered events are applicative events generated by operation invocations on components interfaces and access the components attributes, structural events related to changes in the topology of the considered target system (additions, removals, replacements of components and bindings between components) and system events typically generated by the underlying JVM and OS. Applicative and structural events will be typically detected and notified by interceptors. System events will typically come from monitoring system such as WildCat, Lewys/CLIF, JMX, etc. Conditions relate to the states of the considered system known typically by FPath queries on components attributes and system structure (and possibly behavior). Actions range from simple components attributes settings or external notifications (e-mail, SMS) to complex (possibly transactional) reconfigurations (typically expressed with FScript).

Execution Model The basis of the execution model for component-based systems is the execution unit delimited by the interval between the reception of an operation invocation on a server interface and the emission of a response onto a client interface. Applicative events (generated by operation invocations) and structural events (add, remove of components and bindings) are thus decomposed into two signals *begin* and *end*. Other forms of events (e.g. system events) can be integrated in the model by considering their begin and end signals are merged (i.e. they represent both the same execution point or point in time). The execution model will also typically define event processing modes (instance-oriented or set-oriented triggering of rules) and coupling modes (execution in a immediate, delayed or differed mode in a same or separate thread of execution).

Architectural Integration The reference architecture of the ECA mechanism is based on the concept of management domain. A domain a set of entities on which is applied a common policy. A domain embodies a unit of composition and a unit of control. The reference architecture is hierarchy of nested domains implemented as components: a policy component encapsulates a set of rules components and provide them with a execution strategy in case of multiple or cascading rules, a rule component encapsulates a event component, a condition component and an action component and provide them with a local execution strategy (event processing mode, coupling modes). Event, condition and action components encapsulate sets of applicative components which embody the scope of event detections, conditions evaluations and actions executions.

Summary To summerize, Selfware will propose as part the Selfware architecture an active rule model, i.e. a rule definition model and a rule execution model, that can be coherently integrated into a component model ; and a graceful architecture for the integration of active rules into component-based systems in which the rules as well as their semantics (execution model, behaviour) are represented as components, which permits i) to construct personalized rule-based systems and ii) to modify dynamically the rules and their semantics in the same manner as the underlying component-based system by means of configuration and reconfiguration. These foundations form the basis of a framework/toolkit which can be seen as a library of components to construct events, conditions, actions, rules and policies (and their execution sub-components). The framework is extensible: additional components can be added at will to the library to render more elaborate and more specific semantics according to specific applicative requirements.

6.5.2 Production (deductive) rules

Introduction The underlying idea of a production rule engine is to externalize the business or application logic [4]. A production rule engine can be viewed as a sophisticated interpreter of if-then statements. The if-then statements are the production rules. A production rule is composed of two parts, a condition and an action: When the condition is met, the action is executed. The if portion contains conditions (such as amount ¿=$100), and the then portion contains actions (such as offer discount 5%). The inputs to a rule engine are a collection of rules called a rule execution set and data objects. The outputs are determined by the inputs and may include the original input data objects with modifications, new data objects, and possible side effects (such as sending email to the customer).

Production rule engines should be used for applications with highly dynamic business logic and for applications that allow end users to author business rules. A production rule engine is a great tool for efficient decision making because it can make decisions based on thousands of facts quickly, reliably, and repeatedly.

Adopting a production rule-based approach for your applications has the following advantages:

- Rules that represent policies are easily communicated and understood.
- Rules retain a higher level of independence than conventional programming languages.
- Rules separate knowledge from its implementation logic.
- Rules can be changed without changing source code; thus, there is no need to recompile the application's code.
- Speed and Scalability : The Rete algorithm, Leaps algorithm, and its descendents such as Drools' Reteoo (and Leaps), provide very efficient ways of matching rule patterns to your domain object data. These are especially efficient when you have datasets that do not change entirely (as the rule engine can remember past matches). These algorithms are battle proven.

Which Rule Engine to use? There exists several Production Rule Engine implementations on the market. The following ones were selected to be supported by Selfware because of their popularity, open source and varied target audience:

- JBoss Rules, previously known as Drools [1], which is free, open source, and distributable,
- Jess [2], which is free for personal usage and not distributable,
- Mandarax [5], which is open source.

They all implement the JSR-94 specification, which allows bypassing the vendor lock in. The specification does not encompass the expression language used to define the Rules. Although this means that rules are expressed in files with different formats, the same concept is applied everywhere. Conditions are expressed on properties of Java objects and some Java code is given in case those conditions are met. There is currently some ongoing works to propose a common rule format. Thus W3C is working on the Rule Interchange Format (RIF) and the OMG has started to work on a standard based on RuleML.

JBoss rules is selected for the following advantages :

- is open source (required)
- we are more confortable with this product due to our background acquired during the jimys project (http://forge.objectweb.org/project/download.php?group_id=5&file_id=4795).
- better performance compare to mandarax
- active and dynamic community
- Eclipse plugin for editing the rules

JBoss Rules JBoss rules (former Drools) is a Rule Engine that uses the Rule Based approached to implement an Expert System and is more correctly classified as a Production Rule System. The term "Production Rule" originates from formal grammer - where it is described as "an abstract structure that describes a formal language precisely, i.e., a set of rules that mathematically delineates a (usually infinite) set of finite-length strings over a (usually finite) alphabet" (wikipedia).

JBoss rules has implementations for both Rete and Leaps; Leaps is considered experimental, as it is quite new. The Drools Rete implementation is called ReteOO signifying that Drools has an enhanced and optimised implementation of the Rete algorithm for Object Oriented systems. Other Rete based engines also have marketing terms for their proprietary enhancements to Rete, like RetePlus and Rete III. It is important to understand that names like Rete III are purely marketing where, unlike the original published Rete Algorithm, no details of implementation are published; thus asking a question like "Does Drools implement Rete III?" is nonsensical. The most common enhancements are covered in "Production Matching for Large Learning Systems (Rete/UL)" (1995) by Robert B. Doorenbos.

The Rules are stored in the the Production Memory and the facts that the Inference Engine matches against the Working Memory. Facts are asserted into the Working Memory where they may then be modiied or retracted. A system with a large number of rules and facts may result in many rules being true for the same fact assertion, these rules are said to be in conflict. The Agenda manages the execution order of these conflicuting rules using a Conflict Resolution stategy.

There are two methods of execution for a Production Rule Systems - Forward Chaining and Backward Chaining; systems that implement both are called Hybrid Production Rule Systems. Understanding these two modes of operation are key to understanding why a Production Rule System is different and how to get the best from them. Forward chaing is 'data-driven' and thus reactionary - facts are asserted into the working memory which results in one or more rules being concurrently true and scheduled for execution by the Agenda - we start with a fact, it propagates and we end in a conclusion. Drools is a forward chaining engine.

A rule has the following rough structure:

rule "name" ATTRIBUTES when LHS then RHS end

LHS (Left Hand Side) is the conditional parts of the rule. RHS (Right Hand Side) is basically a block that allows Java semantic code to be executed

6.5.3 Workflow engine

Introduction A workflow engine, sometimes referred as a BPM (Business Process Management) engine, is a software component that breaks a work process down into tasks. A basic example of such a process is an approval workflow process, in which an employee needs a manager's permission before running an application. A workflow engine provides an infrastructure to model this workflow, execute it, assign the tasks to its participants, and monitor it. To achieve the desired results, it may interact with humans or machines through, for example, Web services. This enables integration with platforms different from Java, like mainframes or .NET.

Bonita workflow Bonita is a workflow solution for handing long-running, user-oriented workflows providing out of the box workflow functionalities to handle business processes. Bonita is Open Source and is downloadable under the LGPL License (http://bonita.objectweb.org). Its main key benefits are:

- A comprehensive set of integrated graphical tools for performing the process conception and definition, the instantiation and control of this process, and the interaction with the users and other applications.
- 100% browser-based environment with Web Services integration that uses SOAP and XML Data binding technologies in order to encapsulate existing workflow business methods and publish them as JavaEE-based web services.
- A Third Generation Worflow engine based in the activity anticipation model. This flexibility allows a considerable increase of speed in the design and development phases of cooperative applications.

- Support of the XPDL standard, backed by the WfMC (Workflow Management Coalition).

6.6 Navigation (EMN)

In order to take informed adaptation decisions and apply them, an autonomic framework must have access to the architecture of the target application. Moreover, as this architecture will change dynamically, this information must be obtained directly from the application or from some causaly connected representation of it, and not from static sources like ADL files or deployment descriptors.

More precisely, the Selfware platform needs a way to:

1. introspect the architecture and state of the target application in order to decide which changes to apply, if any. This covers the C (condition) part of decision rules.
2. denote elements of the application to which the reconfigurations it has decided must apply. This covers the A (action) part of the rules.

The Fractal component model chosen for the Selfware platform already supports introspection of all the elements of an architecture. It is thus possible to write programs in a general purpose language (e.g. Java) which navigate in an architecture to locate the elements which must be reconfigured. However, such a direct use of the Fractal API is a not very practical or flexible: it requires writing verbose and error-prone code, and, in the case of Java, adds code compilation and deployment phases.

Instead, we propose to use a special notation, FPath [**?**], designed specifically to express queries on Fractal architectures. These queries can be used to "walk" inside a running Fractal application, discovering its structure and selecting elements according to any property represented in the Fractal model. The FPath notation features: a concise and readable syntax, inspired by the XPath [**?**] language (note that FPath is not based on XPath or XML, it justs borrows XPath's syntaxic structure); supports all the introspection capabilities of the standard fractal model; extensible to support new introspection capabilites introduced by Fractal extensions without changing the syntax. FPath queries can be used, for example, to locate all the compoents in a given architecture which expose configuration attributes, or to find components which implement or require a given interface.

In the rest of this section, we first present the conceptual model used by FPath to represent a Fractal architecture, then we present the syntax and semantics of FPath queries before showing some concrete examples.

6.6.1 Conceptual Model

FPath sees a given Fractal architecture as an oriented graph with labelled arcs (this is a conceptual model, and does not mean an implementation has to create such a graph). Different kinds of nodes represent all the architectural elements we chose to reify:

- the *components* themselves (not reified as such in Fractal, but only through the `component` interface);
- component *interfaces* (both external and internal);

- configuration *attributes*, corresponding to getter/setter methods on `attribute-controllers`;
- and finally individual *methods* on the interfaces.

These nodes are connected by labelled arcs, which denote the kind of relation between them. For example, an arc labelled `interface` goes from a given component node to each interface node representing the component's interfaces. In the same way, if composite C_1 contains C_2 as a sub-component, the corresponding nodes N_1 and N_2 will be connected by two arcs: $N_1 \xrightarrow{\texttt{child}} N_2$ and $N_2 \xrightarrow{\texttt{parent}} N_1$.

FPath provides a default set arc types, called *axes*, which cover all the relationships defined in the standard Fractal model. These axes can be used to to navigate between `component`s ands their `attribute`s and `interface`s, to follow interface `binding`s, and to navigate between components' `children` and `parent`s. FPath also defines a few extended axes which do not map directly to concepts in the Fractal API, for exemple to find a component's `sibling`s, or all its `ancestor`s or `descendant`s (including indirect ones).

6.6.2 Syntax and Semantics of FPath Queries

Given this representation, FPath expressions (queries) denote *relative paths* starting from an initial (set of) node(s) in the graph. Such a path is made of a series of steps, each made of up to three elements: `axis::test[predicate]` (the predicate is optional). On each step, an initial set of nodes is converted to a new set by following all the arcs with a label corresponding to the axis, then filtering the result using the *test* (on the node names) and optional *predicates* (boolean expressions applied to each candidate). For a multi-step path, this algorithm is repeated with the result of the previous step as the current node-set of the next.

For example, the FPath expression `sibling::*/interface::*[provided(.)][not(bound(.))]` is made of two steps. The first one uses the `sibling` axis, an "empty" test `*` (which is always true) and has no predicate. The second step uses the `interface` axis, no test either, and two predicates which are combined. Inside the predicates, the dot "`.`" represents the current node on which the predicate is evaluated. Evaluating the complete expression starting from an initial component node will: *(i)* select all its sibling components, however they are named; *(ii)* select all the external interfaces of these siblings; *(iii)* filter this set of interfaces to return only server interfaces (`provided()`) which are not already bound.

The expressions used as predicates can be any FPath expression, which includes not only paths but also standard arithmetic operations, comparisons, function calls, litteral strings and numbers and finally variable references (`$varName`). When a path expression is used as a predicate, it is considered true if and only if it returns a non-empty set of nodes. For example, to find all the components in a application which provide configuration attributes, one could use the following expression on the application's root component: `descendant-or-self::*[attribute::*]`. This initially selects all the components contained in the root, recursively, and then filters this set to retain only those from which the step `attribute::*` returns a non-empty set, i.e. the nodes which have configuration attributes. Note that this expression is different from `descendant-or-self::*/attribute::*`, which returns the configuration attributes themselves, not the components which provide them.

6.7 Reconfiguration (EMN)

The Fractal component model used in Selfware is fully dynamic and reflexive and makes it possible to program dynamic reconfigurations, even unanticipated ones, to be executed in a running application. This is important in order to evolve applications without stopping and redeploying them (for example to update a component or subsystem). However, as was the case in the previous section with the introspection features, direct use of the Fractal APIs to program reconfigurations has several drawbacks: verbose and error-prone code due to the lack of language integration and minimalist design of the APIs, compilation and code deployment phases which complicate the process (in the case of Java). Using the bare APIs – especially in a general purpose language – also makes it difficult to guarantee the correctness of the reconfigurations: individually correct Fractal reconfigurations can result in globally incorrect reconfiguration depending on when and how they are executed with respect to each other and to the normal execution of the application. If the Selfware platform is to be used to reconfigure applications during their execution, it is essential we guarantee the application will not break.

We can distinguish two separate but related needs for the Selfware platform with respect to reconfiguration. First, we need a specific language (DSL) to describe the reconfigurations. As stated above, direct usage of the APIs in a general purpose language has several issues. A DSL can provide domain-specific notations and abstractions, and because it targets a more limited range of programs, is more amenable to verification and optimization. In the next subsection, we present the FScript Domain-Specific Language, which builds upon the FPath notation and extends it with the ability to specify complex reconfigurations of Fractal architectures. Second, we need to be able to define correctness for these reconfigurations, which might depend on the target application, and to guarantee that only correct reconfigurations can be applied, as efficiently as possible. The second sub-section shows how this can be done using the generic concept of transactions applied at the level of the Fractal model: how the properties of transactions solve this issue and what it takes to implement them in the context of Fractal.

The works presented in these two sections are related but somewhat independant. Although the syntactic aspects of FScript are interesting by themselves, the main benefit of using a DSL is to offer guarantees on the behaviour of the programs, in this case the correctness of the reconfigurations. On the other hand, the support for transactional reconfigurations described in the second sub-section can be used independently of FScript, e.g. from a Java program, but using it through FScript can enable more powerful and/or efficient verifications, as the transactional layer can then make stronger assumptions (for example that the reconfiguration will always terminate, a property of FScript programs). In practice, the current implementation of FScript includes custom support for transactional reconfigurations (with some limitations). This will be replaced by the more general and powerful solution described here when it becomes available, which will itself also be usable independently of FScript.

6.7.1 The FScript Language

FScript is a Domain-Specific Language designed to program structural reconfigurations of Fractal architectures. Compared to the use of the standard Fractal APIs in a general purpose language, FScript offers better syntactic support for navigation, more dynamicity, and guarantees on the consistency of the reconfigurations.

To do this, FScript relies on the FPath notation presented in section 6.6 to navigate intuitively inside an architecture and select parts of it. FScript supplements FPath by adding the possibility to define complex (scripted) reconfiguration actions to architecture elements selected by FPath queries. These elements can be acted upon to reconfigure the architecture using primitive Fractal operations or user-defined reconfigurations scripts. Beyond its direct syntactic support for Fractal concepts, FScript can provide guarantees on the consistency of the reconfigurations by cooperating with a external system which treats Fractal reconfigurations as ACID transactions.

FScript is a simple imperative/procedural language whose main features are its direct syntaxic support for navigation in Fractal architectures thanks to FPath, safety guarantees on the application of the reconfigurations, and a very dynamic implementation where reconfiguration scripts can then be dynamically loaded and executed without an additional compilation phase.

Here is a simple example of the definition of an FScript reconfiguration action which illustrates almost all of FScript constructs. It automatically connects a component's required interfaces by discovering the compatible server interfaces on sibling components.

```
action auto-bind(comp) = {
  // Selects the interfaces to connect
  clients := $comp/interface::*[required(.)][not(bound(.))];
  foreach itf in $clients do {
    // Search for candidates compatible interfaces
    candidates := $comp/sibling::*/interface::*[compatible?($itf, .)];
    if (not(empty?($candidates))) {
      // Connect one of these candidates
      bind($itf, one-of($candidates));
    }
  }
  return empty?($comp/interface::*[required(.)][not(bound(.))]);
}
```

This defines a new reconfiguration action named `auto-bind`. Given a component `comp` as parameter, this action first uses FPath to find all its client interfaces which are not yet bound. The action then iterates over this set of client interfaces: on each iteration, the action searches for compatible interfaces on the siblings of `comp`, again using an FPath query. Finally, the action tests whether this set is empty, and if not, uses the primitive action `bind()` to connect the client interface `itf` to one of the candidates. Finally, it returns a boolean indicating whether all client interfaces have been bound.

FScript provides a standard library of primitive functions and actions which gives the user access to all the information available from the Fractal API, and all the standard reconfigurations, including component instanciation (`new()`), composite content's manipulation (`add()` and `remove()`), connections management (`bind()` and `unbind()`), component lifecycle (`start()` and `stop()`), and configuration (`set-value()` to change component attributes). FScript is designed and implemented so that it is easy to add new primitives corresponding to Fractal extensions.

FScript's design and implementation guarantee some consistency of reconfigurations in part by the language's structure itself, whose expressive power has been limited, and in part by the implementation. More precisely, it guarantees that reconfigurations always terminate (no infinite loops), that they are atomic (i.e. either succesfuly and completely applied or not at all), and that if they succeed, the resulting application will be a valid Fractal architecture. The current implementation of these guarantees is embedded inside the FScript interpreter and has some limitations. The next section describe a more generic and complete solutions to the correctness of reconfigurations which, although independant of FScript is planned to replace FScript's custom solution in the future.

6.7.2 Support for Transactional Reconfigurations

Dynamic reconfigurations allow modifications of a part of a system during its execution without stopping it entirely so as to maximise its availability. Thanks to properties of component models like modularity and loose coupling, reconfigurations can rely on component-based architectures. However, runtime modifications can let the system in an inconsistent state and we identified three main reliability problems when reconfiguring systems:

1. A first problem when modifying a system at runtime is the synchronization between reconfigurations and the functionnal execution of the system. Actually, the part of the system which is modified could be unavailable for functional execution during the reconfiguration time. To take the hotswap example with a stateful component, calls on the old component must be blocked until a "quiescent state" is reached, then the state must be transfered, and finally previous calls are forwarded towards the new component.

2. A second problem at the model level is about consistency violation by reconfigurations. Component models and application models should define what this consistent system is. So we must ensure the conformity of the system to the model and what we call integrity constraints after reconfigurations.

3. The third and last problem is linked to the composition of reconfiguration operations. The semantics of reconfiguration operations implies there can be some conflicts between them in case of compostion and for synchronization between several reconfigurations.

We think that well-defined transactions associated with structural and behavioral constraints verification is a mean to guarantee the reliability of reconfigurations in component models. In our approach, we defined each of the ACID properties in the context of component-based systems:

- **Atomicity**: either the system is reconfigured or it is not. Each reconfiguration operation must specify its reversible operation. Thus if a reconfiguration transaction is rollbacked, it is possible to come back in a previous stable state by undoing operations. Transaction demarcation is either programmed in the language or automatic.

- **Consistency**: a transaction must be a correct transformation of the system state. So the reconfigured application must be conform to the component model and application-specific constraints. A reconfiguration transaction can be commited only if the resulting system respects the constraints. Other faults like software and hardware failures are the responsibility of the commit protocol.

- **Isolation**: several reconfiguration transactions are independant and any schedule of reconfiguration operations must be equivalent to their serialization. The scheduling must respect the operation semantics and conflicts.

- **Durability**: once a reconfiguration completes with success (commit), the new state is persistent. For every transaction, operations are logged in a journal so that reconfigurations can be redone in case of failure. The application state (architecture and component state) is periodically checkpointed so that any component can be recovered in its last stable state resulting from the last successful reconfiguration.

In our proposal, system consistency relies on **integrity constraints** both at the application and at the model level. An integrity constraints is a predicate which concerns the validity of an assembly of architectural elements but it can also concern component state. An example of such a constraint at the component model level is hierarchical integrity (bindings between components must respect the component hierarchy). Constraints must be checked both at compile time on the ADL configuration and at runtime. We represent the Fractal component model as a typed graph and then each fractal-based application is also a graph which is a well-typed instance of the typed graph and is provided at runtime by the reflexivity of the model. The vertexes are elements from the component model (components, interfaces, etc.) and the edges represent relations between the elements (composition links, binding links etc.) Then integrity constraints can be specified on the graphs with a constraint language "à la OCL", basically an extension of FPath with invariants, preconditions, postconditions.

```
// Example of a precondition for removing a component
operation: void removeSubComponent(Component sub);
preconditions:
// all interfaces of the sub-component are unbound
not(exists(sub/interface::*[not(bound(.))]));
```

To compose operations and regardless of a dedicated reconfiguration language, we consider sequences or parallel executions of intercession operations with conditions expressed by means of introspection operations but all compositions are not always valid. We want to make operation semantics explicit in terms of preconditions and postconditions with our constraint language and we want eventually to be able to change it and to specify new primitive operations. We distinguish two types of conflicts between operations:

- **Parallel conflicts**: for two given reconfigurations *R1* and *R2* executed on the same system, a parallel conflict occurs if *R1* and *R2* modify the same manageable elements in the system model (e.g. bind and unbind operations).

- **Execution dependencies**: an execution dependency occurs if *R1* either need *R2* to be executed first (e.g. stop before unbind)or if *R1* cannot be executed after *R2*. That is to say *R2* postconditions cover or not *R1* preconditions.

To deal with reconfiguration concurrency, we propose a pessimistic approach with locking based on operation semantics to avoid inconsistent compositions of operations. We see two different possibilities for the locking algorithm:

- The first one is to lock directly reconfiguration operations: either conflicts between operations are automatically calculated thanks to their preconditions and postconditions or conflict must be explicitely defined.
- The second one is to use a modified DAG locking algorithm on our instance graph defined in. Then the lock granularity is defined by the manageable elements in the graph representation (e.g., a lock acquisition on a component also locks all its interfaces and every operations in each interfaces).

Another approach to locking is to constrain the execution order of reconfiguration operations with a simple language inspired of behavior protocols in [41]. The protocol compliance is checked at runtime by intercepting reconfiguration calls.

References

[1] Drools. JBOSS. http://labs.jboss.com/jbossrules/docs.

[2] Jess. Jess. http://herzberg.ca.sandia.gov/jess/.

[3] Jms. Sun Microsystem. http://java.sun.com/products/jms/.

[4] Jsr94. Sun Microsystem. http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html.

[5] Mandarax. Mandarax. http://mandarax.sourceforge.net/.

[6] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, November 2002.

[7] Apache - HTTP Server Project. Apache. http://httpd.apache.org/.

[8] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, and M. Kalantar. Oceano - SLA based management of a computing utility. In *7th IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[9] M. Aron, P. Druschel, , and W. Zwaenepoel. Cluster Reserves: a mechanism for resource management in cluster-based network servers. In *International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS-2000)*, Sant Clara, CA, June 2000.

[10] Research projects in autonomic computing. IBM Research, 2003. http://www.research.ibm.com/autonomic/research/projects.html.

[11] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *EWSA*, pages 1–17, 2005.

[12] S. Bouchenak, F. Boyer, D. Hagimont, and S. Krakowiak. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS-2005)*, Orlando, FL, October 2005.

[13] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *International Workshop on Component-Oriented Programming (WCOP-02)*, Malaga, Spain, June 2002. http://fractal.objectweb.org.

[14] B. Burke and S. Labourey. Clustering With JBoss 3.0. October 2002. http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html.

[15] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. A Microrebootable System: Design, Implementation, and Evaluation. In *6th Symposium on Operating Systems Design and Implementation (OSDI-2004)*, San Francisco, CA, December 2004.

[16] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *4th ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[17] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference, Freenix track*, Boston, MA, June 2004. http://c-jdbc.objectweb.org/.

[18] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *18th Symposium on Operating Systems Principles (SOSP-2001)*, Chateau Lake Louise, Banff, Canada, October 2001.

[19] Y. Chawathe and E. A. Brewer. System Support for Scalable and Fault-Tolerant Internet Services. In *Distributed System Engineering. The British Computer Society*, 1999.

[20] S. W. Chen, A. C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste. An Architecture for Coordinating Multiple Self-Management Systems. In *4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4)*, Oslo, Norway, June 2004.

[21] J. Dowling and V. Cahill. Self-managed decentralised systems using k-components and collaborative reinforcement learning. In *1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04)*, New York, NY, 2004.

[22] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *4th USENIX Symposium on Internet Technologies and Systems (USITS-2003)*, Seattle, WA, March 2003.

[23] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable. *IEEE Computer*, 37(10), October 2004.

[24] Stella Gatziu, Arne Koschel, G&#252;nter von B&#252;ltzingsloewen, and Hans Fritschi. Unbundling active functionality. *SIGMOD Rec.*, 27(1):35–40, 1998.

[25] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *1st Workshop on Self-Healing Systems (WOSS'02)*, New York, NY, 2002.

[26] X. He and O. Yang. Performance Evaluation of Distributed Web Servers under Commercial Workload. In *Embedded Internet Conference 2000*, San Jose, CA, September 2000.

[27] A. Iyengar, E. MarcNair, and T. Nguyen. An Analysis of Web Server Performance. In *IEEE Global Telecommunications Conference (GLOBECOM'97)*, Phoenix, AR, November 1997.

[28] JOnAS Project. Java Open Application Server (JOnAS): A J2EE Platform. http://jonas:objectweb.org.

[29] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1), 2003.

[30] J. C. Knight, D. Heimbigner, A. Carzaniga Alexander Wolf, J. Hill, P. Devanbu, and M. Gertz. The Willow Survivability Architecture. In *4th Information Survivability Workshop (ISW-2001/2002)*, Vancouver, Canada, March 2002.

[31] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[32] M.Y. Luo and C. S. Yang. Constructing Zero-Loss Web Services. In *20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-2001)*, Anchorage, AL, April 2001.

[33] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. Supporting adaptable distributed systems with formaware. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pages 320–325, Washington, DC, USA, 2004. IEEE Computer Society.

[34] MySQL. MySQL Web Site. http://www.mysql.com/.

[35] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *6th Symposium on Operating System Design and Implementation (OSDI-2004)*, San Francisco, CA, December 2004.

[36] J. Norris, K. Coleman, A. Fox, and G. Candea. OnCall: Defeating Spikes with a Free-Market Application Cluster. In *1st International Conference on Autonomic Computing (ICAC-2004)*, May 2004.

[37] ObjectWeb Open Source Middleware. C-JDBC: Clustered JDBC. http://c-jdbc.objectweb.org/.

[38] Oscar. Oscar - An OSGi framework implementation. http://oscar.objectweb.org/.

[39] OSGi Alliance. The OSGi Service Platform - Dynamic services for networked devices. http://www.osgi.org/.

[40] Norman W. Paton, F. Schneider, and D. Gries, editors. *Active Rules in Database Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[41] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[42] PLB. PLB - A free high-performance load balancer for Unix. http://plb.sunsite.dk/.

[43] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. QoS-Driven Server Migration for Internet Data Centers. In *10th International Workshop on Quality of Service (IWQoS 2002)*, Miami Beach, FL, May 2002.

[44] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-Based Mail Service. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[45] G. Shachor. Tomcat Documentation. The Apache Jakarta Project. http://jakarta.apache.org/tomcat/tomcat-3.3-doc/.

[46] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *5th USENIX Symposium on Operating System Design and Implementation (OSDI-2002)*, December 2002.

[47] S. Sudarshan and R. Piyush. Link Level Load Balancing and Fault Tolerance in NetWare 6. NetWare Cool Solutions Article. March 2002. http://developer.novell.com/research/appnotes/2002/march/03/a020303.pdf.

[48] Sun Microsystems. Java 2 Platform Enterprise Edition (J2EE). http://java.sun.com/j2ee/.

[49] Sun Microsystems. Java DataBase Connection (JDBC). http://java.sun.com/jdbc/.

[50] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[51] B. Urgaonkar and P. Shenoy. Cataclysm: Handling Extreme Overloads in Internet Services. Technical report, Department of Computer Science, University of Massachusetts, November 2004.

[52] B. Urgaonkar and P. Shenoy. Sharc: Managing CPU and network bandwidth in shared clusters. *IEEE Transactions on Parallel and Distributed Systems*, 15(1), 2004.

[53] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic Provisiong of Multi-Tier Internet Applications. In *2nd International Conference on Autonomic Computing (ICAC-2005)*, Seattle, WA, June 2005.

[54] H. Zhu, H. Ti, and Y. Yang. Demand-driven service differentiation in cluster-based network servers. In *20th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM-2001)*, Anchorage, AL, April 2001.