Livrable Selfware SP1

Lot 2, November 27th, 2007

Selfware Common Services for Self-Management

Authors
Fabienne Boyer (INRIA/Sardes)
Thierry Coupaye (France Telecom)
Pierre-Charles David (LINA/EMN)
Bruno Dillenseger (France Telecom)
Daniel Hagimont (IRIT/ENSEEIHT)
Thomas Ledoux (LINA/EMN)
Marc Léger (France Telecom-EMN)

Version 1.0

Contents

1 Introduction

Autonomic computing, which aims at the construction of self-managing and self-adapting computer systems, has emerged as an important goal for many actors in the domain of large scale distributed environments and applications. This approach more precisely aims at providing systems and applications with self-management capabilities, including self-configuration (automatic configuration according to a specified policy), self-optimization (continuous performance monitoring), self-healing (detecting defects and failures, and taking corrective actions), and self-protection (taking preventive measures and defending against malicious attacks).

Following this approach, the Selfware project aims at providing an infrastructure for developing autonomic management software. An important aspect of this infrastructure is the adoption of an *architecture-based* control approach as described in the SP1-L1 document, meaning that the control loops that regulate the system have the ability to introspect the current software architecture of the managed system, as well as they have the ability to modify (i.e. reconfigure) this architecture.

The introspection and reconfiguration capabilities that can be invoked within control loops mainly rely on the *common services* provided by the Selfware infrastructure. These services act on a managed system. Examples of these services are a deployment service, allowing to deploy a managed legacy software in a distributed environment, a monitoring service allowing to observe a given managed element, or a scripting language allowing to dynamically explore the software architecture of a managed system.

The objective of this document is to give details on the common services provided by the Selfware platform. The following sub-section recalls the main design principles of the Selfware platform, and then specifies the list of common services that are provided by this platform.

The software parts associated to the present document are available on the site http://wiki.jasmine.objectweb.org/xwiki/bin/view/Main/WebHome.

2 Selfware Design Principles

This section recalls the main design principles of the Selfware platform, based on the notions of Managed Elements and Autonomic Manager. Then it lists the set of common services that are provided to Managed Elements and Autonomic Managers.

2.1 Managed Elements and Autonomic Managers

As detailed in the SP1-L1 document, the autonomic regulation provided by the Selfware infrastructure on a managed system is based on managed elements (ME) and autonomic managers (AM). A system managed with Selfware is more precisely constituted by a collection of managed elements, that may consist of a single elementary hardware of software element, or may be a complex system in itself, such as a clustered application server. A managed element provides sensor and actuator interfaces respectively allowing to observe and manipulate it. Sensor and actuator interfaces are used by autonomic managers, that regulate a managed system through feedback control loops. An autonomic element is the ensemble including a set of managed elements controlled by autonomic managers.

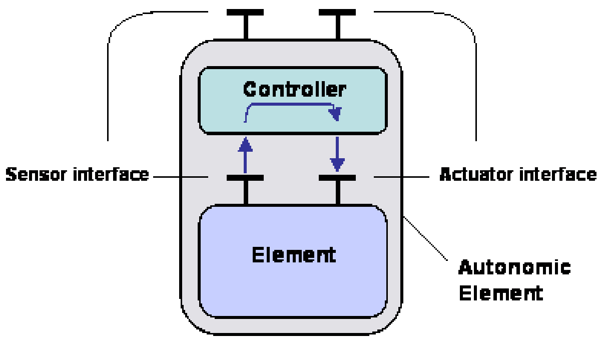


figure 1. An autonomic element

A main design choice is to rely on a component model for building both Managed Elements and Autonomic Managers. The component model we use is Fractal [3]. A managed element is implemented as a Fractal component that encapsulates a controlled legacy entity. In the same way, an autonomic manager is a Fractal component that monitors a set of managed elements, analyzes notifications coming from managed elements sensors, diagnoses the state of the system, decides on a plan of actions and finally, executes the corresponding command plan.

2.2 Common Services

The common services 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. Common services are mainly built themselves with components, which allows to consider them as managed elements in the managed system. The main common services that are currently provided by the Selfware infrastructure are:

- The wrapping service, that allows generating the wrappers used to control legacy software.
- The navigation service designed to express queries on the managed system’s architecture.
- The reconfiguration service, used to define consistent reconfiguration on the managed system’s architecture.
- The resource allocation service, that allows allocating resources (e.g. nodes) for the managed system as well as for the management system.
- The deployment service, that aims at deploying both the managed system and the management system on remote nodes.
- The monitoring service used to gather information on the managed system and to aggregate these information to provide high-level events attached to more semantic.

- The system representation service, used to add reliability to the configuration actions performed by the Autonomic Managers by replicating the critical components.
- The decision service used to implement the reactive part of autonomic managers.

3 Organization of the document

The rest of the document is organized as follows. Section 4 describe the main services that are associated with Managed Elements, allowing to either to build, deploy, configure, or monitor them. Section 5 focuses on the services that are more specifically associated with Autonomic Managers, and puts an emphasis on the rule decision service. It should be noted that, as an Autonomic Manager is also considered as a Managed Element, the services associated with Managed Elements also apply for Autonomic Managers.

4 Services associated with managed elements

4.1 Building Managed Elements

4.1.1 Wrapping principles

Component-based management aims at providing a uniform view of an environment composed of different types of software. Each managed software is encapsulated in a component and the overall environment is abstracted as a component architecture. Therefore, deploying, configuring and reconfiguring the environment is achieved by using the services associated with the used component-based middleware.

The component model we used in Selfware is the Fractal component model [3]. A Fractal component is a run-time entity that is encapsulated and has one or more interfaces (access points 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. The signatures of both kinds of interface can be described by a standard Java interface declaration, with an additional role indication (server or client). Components can be assembled to form a component architecture by binding components interfaces (different types of bindings exists, including local bindings and distributed RMI-like bindings). An Architecture Description Language (XML based language) allows describing an architecture and an ADL interpreter can be used to deploy such an architecture. Finally, Fractal provides a rich set of control interfaces for introspecting (observing) and reconfiguring a deployed architecture.

Any software managed with Selfware is wrapped in a Fractal component which interfaces its administration procedures. Therefore, the Fractal component model is used to implement a management layer (figure 2) on top of the legacy layer (composed of the actual managed software). In the management layer, all components provide a management interface for the encapsulated software, and the corresponding implementation (the wrapper) is specific to each software (e.g. the Apache web server in the case of J2EE). Fractal's control interfaces allow managing the element's attributes and bindings with other components, and the management interface of each component allows controlling its internal configuration state. 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.

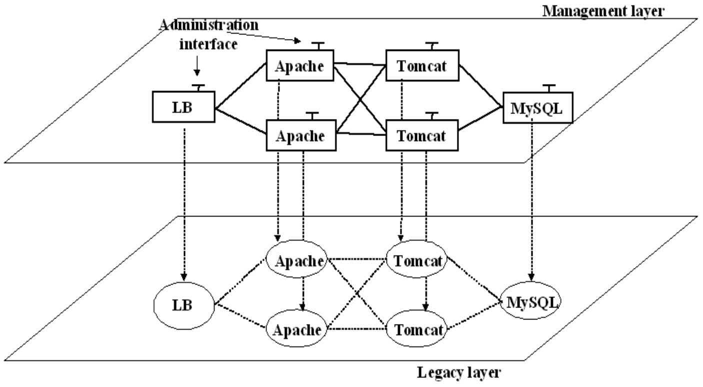


figure 2. Management layer

Here, we distinguish two important roles:

1. The role of the management and control interfaces is to provide a means for configuring components and bindings between components. It includes methods for navigating in the component-based management layer or modifying it to implement reconfigurations.

2. The role of the wrappers is to reflect changes in the management layer onto the legacy layer. The implementation of a wrapper for a specific software may also have to navigate in the component management layer, to access key attributes of the components and generate legacy software configuration files. For instance, the configuration of an Apache server requires to know the location of the Tomcat servers it is bound to.

However wrapping components are difficult to implement. The developer needs to have a good understanding of the component model we use (Fractal), more precisely the programming interfaces of the Fractal component model. Moreover, the implementation of a wrapping component is quite systematic and should be simplified.

Our approach to this problem is to introduce a Wrapping Description Language which is used to specify the behavior of wrappers. A WDL specification is interpreted by a generic wrapper Fractal component, the specification and the interpreter implementing the wrapper. Therefore, the implementation of a wrapping component is much simplified.

4.1.2 Wrapping Description language

A WDL description defines a set of *externs* which correspond to client interfaces of the wrapper component, which can be bound with other components.

It also defines a set of methods that can be invoked to configure or reconfigure the wrapped software. Generally, a WDL specification provides *start* and *stop* operations for controlling the activity of the software, and a *configure* operation for reflecting the values of the component's attributes in the configuration files of the software. Other operations can be defined according to the specific management requirements of the wrapped software. These methods are implemented in Java and can be reused in many cases.

The attributes of the wrapper component don't have to be declared in the WDL description, since the generic wrapper provides a generic interface for the management of attributes.

An example of a WDL specification which wraps an Apache server in a J2EE infrastructure is given hereafter. It defines an extern (*workers*) which generates the definition of a client interface that can be used to connect the Apache server with some Tomcat servers.

It defines *start* and *stop* methods which can be invoked to launch/stop the deployed Apache software, a *configure_apache* method which reflects configuration attributes in the *httpd.conf* Apache configuration file, and a *configure_workers* method which implements the bindings with Tomcat servers in the *workers.properties* Apache configuration file. Each method definition specifies the Java class and method which implements it. The Java implementations of these methods are often generic (e.g. *ConfigurePlainText* which managed an <attribute, value> configuration file) and have been used in the wrapper definitions of many of the software we wrapped (we also had to add an implementation of a configuration method for XML configuration files). A method definition includes the description of the parameters that should be passed when the method implementation is invoked. These parameters may be String constants, attribute values (attributes from the defined wrapper component) or combinaison of both (String expressions).

In the WDL specification below, the start method takes as parameters the shell command that launch the server:

- *dirLocal* is an attribute of the wrapper component and defines the directory where the software is actually deployed on the target machine
- *$dirLocal/bin/httpd* is the name of the binary to be launched
- *$dirLocal/conf/httpd.conf* is the name of the configuration file which is passed to the binary and which is generated by the *configure_apache* method of the wrapper

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<wrapper name='apache'>
    <extern name="workers"/>
```

```
        <method name="start" key="appli.wrapper.util.GenericStart"
                     method="start_with_pid_linux" >
               <param value="$dirLocal/bin/httpd -f $dirLocal/conf/httpd.conf  "/>
        </method>
        <method name="configure_apache" key="appli.wrapper.util.ConfigurePlainText"
                     method="configure">
               <param value="$dirLocal/conf/httpd.conf"/>
               <param value="ServerRoot:$dirLocal" />
               <param value="Port:$port"/>
               <param value="User:$user"/>
               <param value="Group:$group"/>
               <param value="ServerAdmin:$serverAdmin"/>
        </method>
        <method name="configure_workers" key="appli.wrapper.util.ConfigureWorkers"
                     method="configure">
               <param value="$workers.nodeName"/>
               <param value="$workers.workerPort"/>
        </method>
        <method name="stop" key="appli.wrapper.util.GenericStop"
               method="stop_with_pid_linux" >
        </method>
    </wrapper>
```

figure 3.A WDL specification

The *configure_apache* method is implemented by the *ConfigurePlainText* Java class. This configuration method generates a configuration file composed of <attribute,value> pairs:

- *$dirLocal/conf/httpd.conf* is the name of the configuration file to generate
- the attributes and values are separated by a ":" character

The *configure_workers* method is implemented by the specific *ConfigureWorkers* Java class. This configuration method generates a configuration file which describes the bindings between Apache and Tomcat servers. In this method, it is necessary to navigate in the deployed component architecture to find the Tomcat servers the Apache software is bound with:

- *$workers.nodeName* returns the names of the nodes hosting a Tomcat server which the Apache server is bound to (a list of names separated by commas is returned). *workers* corresponds to a Fractal client interface which references the Tomcat servers (their wrappers). *nodeName* is an attribute defined in all the wrappers, which gives the machine hosting the software.
- $workers.workerPort similarly returns the ports on which the Tomcat servers are receiving requests.

4.2 Deploying Managed Elements

This section describes the deployment service provided in Selfware, including the architecture of the deployment system, the deployment process and the bootstrapping of the platform.

4.2.1 Deployment System Architecture

Deployment in Selfware is architecture-based. It means that given a description of the software architecture, Selfware 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 on Selfware's target nodes, components can also be replaced with new versions. The following figure presents the general architecture of the Selfware deployment system which is composed of four principal elements: (1) The configuration and deployment description, (2) The deployment engine, (3) The targets and (4) The package repository.

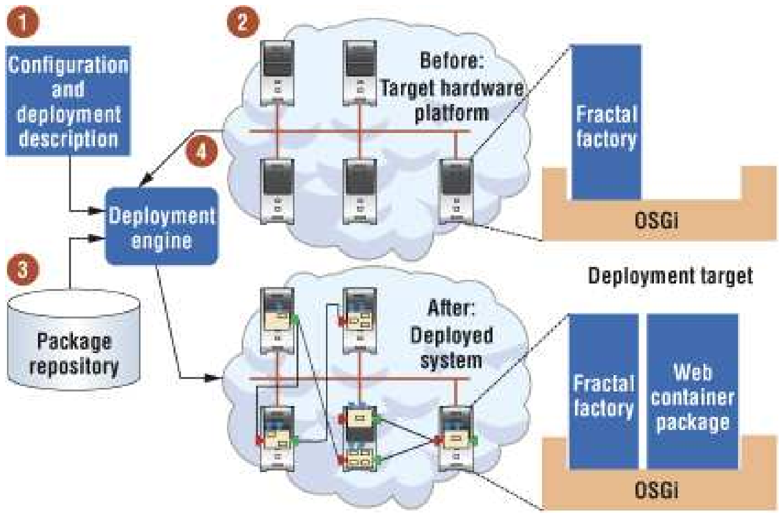


figure 4.The Selfware Deployment System

Several entities of the Selfware deployment system can be mapped, in terms of functionality, to the abstractions defined by the OMG Deployment and Configuration specification. Below we describe in more details all of those elements.

4.2.2 Configuration and Deployment Description

Configuration and deployment description is an input for the deployment system. It contains all the information needed by the deployment system to deploy a given Selfware-enabled application. A minimum set of such information is the following:

- Architecture of the application to be deployed i.e. components and their relation in terms of hierarchy and interconnections
- Configuration of the components --- values of their attributes
- Placement information, i.e. on which machine which component is to be deployed, or certain constraints on component co-location, without explicit information on the target nodes
- Packaging information, i.e. in which software package contains the code and other resources needed by given component

At present there are three ways to describe a deployment configuration for Selfware --- via the Architecture Description Language, via BeanShell commands or using FScript.

ADL

Selfware Architecture Description Language (ADL) is an extension of the Fractal ADL. Therefore, it is XML-based and provides a static description of the system we want to deploy. Below is an example of a Selfware deployment file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
  "classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<!-- ================================ -->
<!--       J2EE ARCHITECTURE          -->
<!-- ================================ -->
<definition name="J2EE">
  <interface name="service" ole="server" signature="fr.jade.service…/>

  <!--              START              -->
  <component name="start" definition="fr.jade.resource.start.StartType">
    <virtual-node name="node1" />
  </component>

  <!--              APACHE             -->
  <component name="apache"
          definition="fr.jade.resource.j2ee.apache.ApacheResourceType">
    <attributes
```

```
             signature="fr.jade.fractal.api.control.GenericAttributeController">
            <attribute name="resourceName" value="apache" />
            <attribute name="dirLocal" value="/tmp/j2ee" />
            <attribute name="user" value="jlegrand" />
            <attribute name="group" value="jlegrand" />
            <attribute name="port" value="8081" />
            <attribute name="serverAdmin" value="julien.legrand@inrialpes.fr"/>
            <attribute name="jkMounts" value="servlet" />
          </attributes>
          <virtual-node name="node1" />
          <package name="Apache Wrapper" />
        </component>

        <!--                TOMCAT              -->
        <component name="tomcat"
          definition="fr.jade.resource.j2ee.tomcat.TomcatResourceType">
          …
          <package name="Tomcat Wrapper" />
        </component>

        <!--                MYSQL               -->
        <component name="mysql"
          definition="fr.jade.resource.j2ee.mysql.MysqlResourceType">
          …
        </component>

        <!--              BINDING               -->
        <binding client="this.service" server="start.service" />
        <binding client="apache.worker" server="tomcat.resource" />
        <binding client="tomcat.jdbc"   server="mysql.resource" />
        <binding client="start.rsrc_mysql"   server="mysql.resource" />
        <binding client="start.rsrc_tomcat" server="tomcat.resource" />
        <binding client="start.rsrc_apache"  server="apache.resource" />
        <virtual-node name="node1" />
      </definition>
```

figure 5.A simple ADL description of a J2EE architecture

The description above defines a simple 3-tier J2EE architecture, which is built by Apache, Tomcat and MySQL servers. As specified by the virtual-node tag, each of the tiers should be deployed on a separate target machine. The virtual-node tag provides only collocation information, i.e. it does not provide information on the exact name or IP address of the target machine, but only says which components should be placed together, and which should not.

The only "dynamic" aspect of this description is the order in which the tiers are started. MySQL needs to be started before Tomcat, which in turn needs to be launched before the Apache server. Since Fractal by default does not allow to specify the order in which components are started, Selfware uses a specific component, called start, to achieve this goal. The start component launches all the components bound to it in an order equal to the one of bindings. Therefore, in the example above, starter will first launch MySQL, then Tomcat and finally Apache.

Information about component packages is provided through the package XML element. Each component specifies zero or one package elements, which are String package identifiers. Depending on the implementation of the package repository from which the packages are obtained, package identifiers can have different forms. At present we reuse the identifiers from OSGi Bundle Repository (OBR), as will be explained later in this chapter. The rest of Selfware ADL is the standard elements found in Fractal ADL.

Scripts

Selfware supports not only ADL-based but also script-based deployment through FScript. Using scripts and an interactive console is interesting especially when a human administrator needs to introspect or modify the architecture of a component-based application. ADL-based deployment, on the other hand, is more adapted for the initial deployment of the system.

4.2.3 Deployment Engine (Selfware Boot)

The deployment engine is an application which given the deployment description file or FScript command as an input is capable to install, instantiate, configure and start or update components. In the current implementation, the deployment engine is a largely modified Fractal ADL factory that consists of three composite components: (1) loader, (2) compiler and (3) backend each of them containing a set of primitive components. The loader composite component is responsible for verifying the correctness of the Selfware deployment file. Compiler component creates tasks that are executed by the backend component. The compiler component consists of several primitive components, which are executed in the top-down order, therefore the package compiler for example is executed before the type compiler.

Parsing of the ADL or the interpretation of the FScript commands is performed on a single machine, called Selfware Boot. This is because it does not seem interesting to have a distributed application for this task. However, once the deployment information has been parsed, the rest of the deployment process is distributed -- the communication between Selfware Boot and target machines (called Selfware Nodes) is performed over a specific, component based implementation of RMI, called the FractalRMI.

4.2.4 Targets (Selfware Nodes)

Targets, also called Selfware Nodes, are machines on which the components can be deployed. Every Selfware Node contains a Fractal factory which is remotely used by Selfware Boot to instantiate components. The most important functionality provided by the targets is support for component installation, instantiation, versioning, updates and removal.

To provide this functionality in the current implementation of the Selfware deployment system, target nodes use OSGi. OSGi provides a layer between a Java Virtual Machine (JVM) and a Java-based application. This layer manages how software modules (called bundles) are installed in the file system and how they are loaded by Java class loaders. In our work OSGi [10] provides the functionality that we require in terms of component versioning and dynamic updates. It also gives us a standard packaging format and a simple package repository, in which component packages are stored, from which they are downloaded in order to be installed on Selfware Nodes, and where inter-package dependencies are resolved. OSGi packages, called bundles are jar files with a specific manifest files, which allows bundles to specify import/export dependencies. Those dependencies exist when two bundles need to use same classes. Since each bundle has its own class loader, without import/export declarations ClassCastExceptions would occur anytime one bundle would try to use code from another bundle.

Selfware currently uses Felix as an implementation of the OSGi specification. Felix implements the OSGi R4 specification, thus the most recent one at the time of writing of this document.

4.2.5 Deployment

Once the Selfware Boot is given a description of the application to be deployed, it starts parsing it. First it verifies that the description is correct in terms of, for example, compatibility of component interfaces. It also verifies that there is sufficient amount of Selfware Nodes available for allocation. If any of these verifications fail, the whole deployment process fails.

If the verification is correct, Selfware Boot creates tasks, responsible for the allocation of nodes, installation of software packages, creation of the type of components, instantiation of components, binding them etc. Clearly those tasks have dependencies in terms of order of execution --- nodes need to be allocated before software packages can be installed, installation has to be performed before component types can be created etc.

Once the tasks have been created and scheduled, they are executed. Here is a description of some of the tasks, in their order of precedence.

- Virtual Node Creation Task: creates a virtual node.
- Install Package Task: installs a given software package on a given virtual node.

- Create Component Task: creates a Fractal component on a given target machine, once the code needed to do it was installed.

4.2.6 Component Updates and Undeployment

Updates and undeployment of application components are important mechanisms that the targets have to provide to application managers. Components may need to be updated because they are buggy, their performance is insufficient or simply a new version of the component's code was released. Similarly, components may need to be undeployed because they are no longer needed or because the system adapts itself to a decreased load. These operations need to be performed "cleanly", namely the undeployed component should be completely removed from the target machine.

To describe how Selfware handles component updates, we assume that the management software initiates the reconfiguration. We assume that a manager has a reference to a component that it wants to redeploy. Moreover, the manager knows about the new package containing this component code. For example, it received a notification about the new version of this package available in the package repository. Consider a composite component C1 composed of two primitives C2 and C3, both bound at runtime. The reconfiguration steps are as follows:

- The manager calls the Component update(ComponentDescription newComponentDesc, PackageId newPackageId) method on the C3 component, passing the component description and the new component package ID as an argument. The ComponentDescription in our Fractal-based implementation is the Fractal component type and the name of the component implementation class.
- This results in C3 contacting the Generic Factory that created it and calling this factory.update(ComponentId self, PackageId newCodePackage) method.
- The Generic Factory deploys the new component, generates a componentId, and returns it to the manager.
- The manager unbinds the old component from the other components and removes it by invoking the remove(componentId) method on C1, the composite. C1 forwards this method call to the Generic Factory that created it.
- The Generic Factory maintains a mapping between each componentId and its packageId. Every component maps to exactly one package. The Generic Factory obtains the component old packageId to remove it and invokes the markUnused(packageId) method on the Installer component.
- The manager now adds the new component C3 (with the new implementation) to the C1 composite, binding the new version of C3 to other components and managing its life cycle.

4.3 Allocation of nodes to Managed Elements

Deploying a component implies the allocation of a node satisfying the component’s constraints in terms of resources requirements. The allocation service may take into account several kinds of properties like the memory size of the hosting node, its CPU speed, etc. Locality constraints between different nodes may also be taken into account in order to place some components close to each other. In the current Selfware infrastructure, the Allocation service only takes into account closeness constraints. It is based on the virtual-node property described in the ADL file of the application to manage.

The Allocation Service has the knowledge, at any instant, of the nodes that are used (i.e., that are hosting some part of either the managed application or the managed system itself) or free. When a new component has to be deployed on a given virtual node that is not already associated to a real node, the Allocation service searches for a free node, and then registers the association between the virtual node and the real node. A node may return to the free state when no more management or application’s part is running on it.

Any node may join the Selfware infrastructure at any time, by getting the location of the Selfware Boot node, and then calling a registration method on this node. In contrast, a node is not supposed to leave the Selfware infrastructure at any time, but only when it is in the free state. Finally, unanticipated leavings of nodes are managed as hardware faults, by the Repair service.

4.4 Navigation and Configuration of Managed Elements

4.4.1 Objectives

Selfware's targeted applications include legacy systems, which by definition present a large variety of forms and functionalities. In order to keep the core of the Selfware platform independent of specific technologies, the managed elements themselves all present the same uniform interface to the platform. Concretely, Fractal wrappers are built using the techniques presented in the previous section. These wrappers expose the architecture and reconfiguration capabilities of the legacy systems in terms of the Fractal component model, providing a powerful and uniform management interface to be uses by the rest of the platform.

For example, an Apache HTTP server, although implemented in C and configured using text files, is seen by the Selfware platform as a normal Fractal component, exposing its lifecycle and configuration attributes through Fractal's standard interfaces (lifecycle and attribute controllers in this case).

Once this wrapping is done, the Selfware platform can manipulate the managed elements using the standard Fractal API. However, the relatively low-level nature of these APIs make it complex to write reconfigurations: the resulting code is often very verbose and mixes high-level concerns with lower, language-level « plumbing »[1], making it difficult to write and to understand. Because even simple reconfigurations can be so cumbersome to write, it is even more difficult to write reliable reconfigurations, which handle appropriately all the corner cases and deal with unexpected errors. In our context, the reliability of the reconfigurations is primordial, as they will be initiated automatically by the platform, without human supervision. For administrators to leave the control of their applications to an automated system, they must have the assurance that whatever it does it will not "break" the managed system by putting it in an unusable state.

4.4.2 Manipulating Fractal Components using FPath and FScript

To overcome the difficulty of writing reliable Fractal reconfigurations, Selfware uses a Domain-Specific Language (DSL) named FScript [5]. By focusing on a limited domain (introspection and reconfiguration of Fractal architectures), FScript can provide better language-level support for Fractal-specific concepts. In addition, we control the language's power of expression, semantics and implementation, which make it possible to provide strong guarantees on the reliability of the reconfigurations. Concretely, FScript is made of two parts:

- FPath**,** a DSL [9] for querying Fractal architectures. Its domain is restricted to the *introspection* of architectures, navigating inside them to locate elements of interest by their properties or location in the architecture. This focused domain allows FPath to offer a concise yet powerful and readable syntax inspired by XPath [15]. FPath sees a Fractal architecture as a directed graph. Nodes in the graph represent components, interfaces and attributes. Directed arcs connect these nodes and indicate the relationship between the corresponding elements in the architecture. For example, a node representing a composite component will be connected in the graph by an arc named “child” to the nodes representing its direct sub-component (and vice-versa with an arc named “parent”). Given this representation, an FPath query “walks” in the graph along the relations represented by the named arcs to select the appropriate architectural elements. FPath can be used by itself, without the rest of FScript, as a general navigation and query language for Fractal.

[1] The Fractal model introduces new concepts like components and interfaces, but most actual implementations, including all the ones in Java, do not extend the host language with appropriate constructs. The typical Java code using the raw Fractal APIs is full of downcasts and low-level objects and arrays manipulations.

- FScript itself allows for the definition of complex reconfigurations of Fractal architectures. FScript integrates FPath seamlessly in its syntax, FPath queries being used to select the elements to reconfigure. By design, FScript is restricted to manipulating the architecture of Fractal systems (structure, state and configuration), and nothing else. For example it has no support for invoking the service interfaces of components (although these are visible). This restricted power ensures that FScript programs can not execute ``dangerous" and difficult to control code constructs (infinite loops, I/O, etc.) as would be the case in a general-purpose scripting language like BeanShell for example. One of the main contributions of FScript is to guarantee the reliability of dynamic, distributed and concurrent reconfigurations in Fractal.To do this, FScript considers complex reconfigurations as *transactions*. The FScript interpreter integrates with a transactional monitor which gives a transactional semantics to the reconfigurations, following the standard ACID properties (Atomicity, Consistency, Isolation and Durability). Indeed, reconfigurations may be invalid and leave a system in an inconsistent state, i.e. no more available/usable from a functional point of view. The execution backend automatically and transparently detects and corrects errors to make the system fault-tolerant during reconfigurations. The ACID properties are unifying concepts of transactions for distributed computation used for supporting concurrency, recovery, and guaranteeing system consistency. To benefit from these properties, each top-level FScript action and function is executed as a separated transaction which can be rolled back in case of failure so that the system comes back in a consistent state.

4.4.3 Integration in the Selfware platform

FPath and FScript are used in the following ways in the Selfware platform (more specifically in Jade).

FPath is used through its Java API in the implementation of the Jade platform itself and in the JORAM wrapper components. Both the Jade platform and the Fractal components wrapping JORAM make heavy use of Fractal's introspection features. Writing the required queries directly in Java can result in complicated and sometimes brittle code, which can often be replaced by a single, one-line FPath query (which is both more readable and robust). Before FPath was used, the original Java code sometimes published directly in a shared registry the references to some components to avoid writing the code required to find them again at some other point. Using FPath made is possible to remove these references (basically global variables), keeping the global registry cleaner. For example, here is the original code that finds all the JORAM servers deployed, written in pure Java:

```
private LinkedList<Component> getAllServers(){
  LinkedList<Component> res = new LinkedList<Component>();
  NamingService ns= Registry.getRegistry(System.getProperty("registry.host"),

  Integer.parseInt(System.getProperty("registry.port")),
                      this.getClass().getClassLoader());
  for (String ref : ns.list()) {
    if(ref.startsWith("JoramServer_")) {
      Component server = ns.lookup(ref);
      try {
        server.getFcInterface("JoramServer-controller");
        res.add(server);
      } catch (NoSuchInterfaceException ignored) { /* Not a server */ }
    }
  }
  return res;
}
```

This code assumes that all the servers are published in the Fractal RMI registry, under a name which matches their Fractal name. The same code rewritten to use FPath looks like this:

```
// Custom FPath function defined in a separate file
function is-joram-server(c) {
 return starts-with(name($c),
          "JoramServer_") && $c/interface::JoramServer-controller];
}
```

```
// Java code
private LinkedList<Component> getAllServers() {
  LinkedList<Component> res = new LinkedList<Component>();
  Node start = fscript.createComponentNode(myself);
  String query = "./parent::*/sibling-or-self::*[is-joram-server(.)]";
  for (Node node : (Set<Node>) fscript.evaluateFrom(query, start)) {
    res.add(((NodeImpl) node).getComponent());
  }
  return res;
}
```

The resulting code is shorter, more readable, and does not rely on Fractal RMI to find the components.

FScript is used to program reconfigurations of the architecture, including the deployment of components in Jade Nodes, in place of ad-hoc BeanShell scripts used earlier. Once a Jade instance is running (a JadeBoot and several JadeNodes), the interactive FScript console can be used to interact with the remote components using Fractal RMI. The console can then be used to navigate inside the platform and managed elements (using FPath queries), and to invoke programmed reconfigurations of their architectures (using FScript scripts). The same FScript scripts can also be invoked automatically as part of an autonomic response of the Selfware platform. Previously, Jade used BeanShell[2] scripts for this. Although BeanShell is more lightweight and dynamic than Java, it is still a general purpose language, with no direct support for Fractal-specific concepts, and none of the guarantees offered by FScript regarding the reliability of the reconfigurations. Because both BeanShell and FScript interact with the platform through Fractal RMI, they can be used at the same time, which is especially important to provide an incremental migration path while BeanShell scripts are converted into FScript.

Other possible uses which are being investigated include writing the autonomic repair algorithm which is integrated in Jade using FScript instead of Java. It is not yet clear whether FScript's power of expression is sufficient to express the algorithm. Also, the interactions between the transactional semantics offered by FScript and the repair feature must be investigated further: both features' goal is to ensure the continuity of service of the managed application(s), but they are triggered by different kinds of errors and use different (but related) techniques.

4.5 Monitoring Managed Elements

4.5.1 Monitoring objectives

With regard to the control loop principle the Selfware architecture is based on, the monitoring service is in charge of getting data from ad hoc sensors associated to the Managed Elements, and to make these data available for the decision function, namely the Autonomic Managers. These data typically describe the dynamic state of the managed element rather than its static constitution (e.g. for a computer, number of processors or memory size). However, changes may occur even to something that would look like a "static constitution". For instance, some advanced computers may have a varying number of processors and memory size. Such changes may be of interest for the autonomic management features, and shall be taken into account by the monitoring service. There are actually two kinds of data:

- plain measures of resource consumption (e.g. CPU time, free memory, database connection pool usage, request queue size in an arbitrary middleware...);
- alarms that notify the occurrence of an event that is not necessarily measurable (e.g. a garbage collector occurrence in a JVM, a node failure, etc.).

The monitoring feature shall be able to use both a push and pull model. The pull model is useful to measure resource consumption at an arbitrary rate while the push model enables quickly delivering alarms. Care should be taken not to overload the transport layer towards the decision function, or

[2] BeanShell is a general-purpose scripting language for Java. It is very close to Java, but supports a more interactive and dynamic usage.

overwhelm the decision function itself, by a huge flow of measures and alarms. For this reason, the monitoring service shall be able to provide statistically condensed measures. However, the decision function may be interested in getting detailed information and accessing to all measures in a given time frame or from a given starting date. So, the monitoring service shall also provide a dynamically enabled or disabled memory. Since the monitoring service is supposed to be on during all the lifetime of the autonomic system, this memory feature shall be enabled in a *reasonable* way so that the corresponding amount of storing space does not endlessly grow.

The monitoring service relies on components that observe a given resource, namely probes. In the following, we describe the architectural description of these probes in two steps:

- basic probes that provide the monitoring service;
- an extension of the basic probes to introduce probe sharing and aggregation through a composite probe architecture.

4.5.2 Probe components

Basically, a probe is a component with an autonomous activity for observing and getting measures from the resource it observes. This activity is controlled accordingly to the lifecycle specification depicted in figure 6.

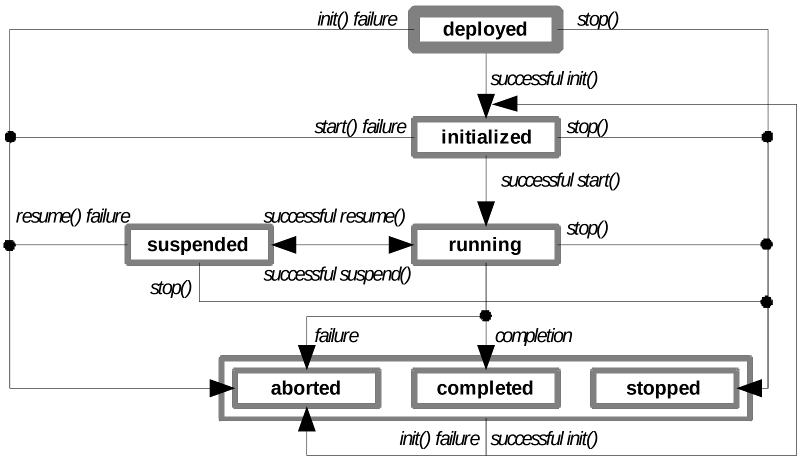


figure 6. Lifecycle of probe components

A probe component first exists in the *deployed* state. It is typically initialized and started, and then possibly suspended and resumed. The end of activity is depicted as a pseudo state that actually represents three states:

- *aborted* means something did wrong and the probe could not achieve what it was supposed to do (i.e. either its computation or a lifecycle transition request);
- *completed* means the probe normally terminated its activity;
- *stopped* means that the blade did not reach the end of its activity, but simply conformed to the *stop* lifecycle transition request.

Once the end of activity has been reached, the blade activity may be rerun after an initialization step. Suspend and resume requests may be useful when some faults have been detected or some reconfiguration is under way, in order to avoid getting meaningless measures and possibly bursty generations of alarms. Suspending a probe is also a way to check the disturbance caused by its activity.

We now go into the details of the basic probe component architecture using the Fractal model. This architecture comes from the CLIF [7] load testing framework's so-called *blade* architecture, hence the frequent use of "blade" in the terminology. As shown by 0, the probe component type consists of three mandatory server interfaces (namely Data_collector_administration, Storage_proxy_administration and Blade_control) and one mandatory client interface (Supervisor_information).

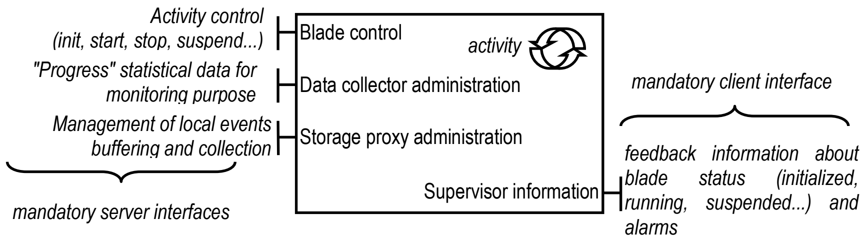

Activity control (init, start, stop, suspend...)
"Progress" statistical data for monitoring purpose
Management of local events buffering and collection
mandatory server interfaces
Blade control
activity
Data collector administration
Storage proxy administration
Supervisor information
mandatory client interface
feedback information about blade status (initialized, running, suspended...) and alarms

figure 7. Fractal-based architecture of basic probes. The probe's activity consists in getting information from the resource it is observing.

Interfaces Blade_control and Supervisor_information are tightly coupled because most of probe activity control operations (init, start, stop, suspend...) are asynchronous: the call returns as soon as the operation processing starts. Once the operation is terminated, a call-back operation from interface Supervisor_information is used to inform the supervisor component about the actual probe state. The reason for asynchronous operations in probe activity control is that we consider scalability issues. A typical usage of activity control operations is to simultaneously initialize, start, suspend, etc. a whole set of probes. We could implement asynchrony at the supervisor's side, simply by using parallel threads calling activity control operations and waiting for operation return. But, first, this could introduce a possible high overload on the supervisor if you consider large scale systems (hundreds of probes or more). Second, we still need a call-back operation to give feedback about the probe state at least for states aborted and completed. As a result, we'd rather introduce asynchronous operations and a unified way of providing the supervisor with feedback information about probes states. At last, interface Supervisor_information provides an operation to notify arbitrary alarm events to the Supervisor. Interface Blade_control offers two extra operations, respectively to consult and modify specific properties. These properties include the activation or deactivation of the memory of the various events (measures, lifecycle, alarms) it generates.

Interface Data_collector_administration provides statistical data about the probe – typically about the measures obtained from the resource it observes. These data are represented as an array of integer values. It may look like an arbitrary limitation not to be able to deliver other data types, but it is actually a pragmatic choice that is directly inspired by the LeWYS project [4]. This choice seems particularly relevant to monitor such things like CPU usage percentage, free memory, average throughputs and response times, etc. In a general way, we consider that for other needs than numerical monitoring resource usage, alarms are a good way of notifying probe events holding data of arbitrary type. For instance, a node failure would be typically notified through an alarm.

Interface Storage_proxy_administration is bound to the storage proxy role played by the probe, to enable possible buffering and final collection of probe events. This interface provides methods to possibly allocate a buffer for a new run, and to collect events.

4.5.3 Composite probes

The basic probes described above are primarily designed to be used in a single level, as a flat layer: each probe is managed one by one and monitors one resource. For both scalability and convenience reasons, it appears useful to be able to compose these basic probes into composite probes whose data don't come from a resource observation, but from a set of other probes, whatever they are basic or composite (see figure 8). Here, the idea is to take advantage on the Fractal model's support for component hierarchy and sharing to be able to:

- obtain as many measures as possible from a minimal set of basic probes, with an adaptable level of details and different aggregated values;
- transparently manage a whole hierarchy of probes through a single composite probe.

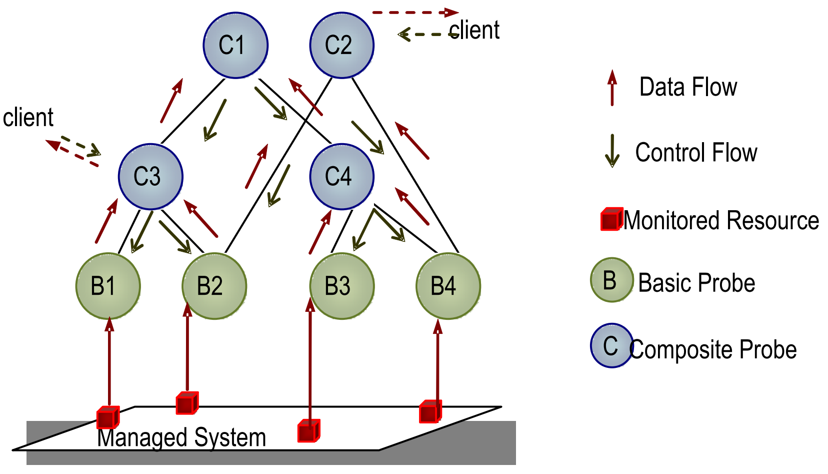


figure 8. General view of a composite probe hierarchy.

Let's take the use case of monitoring the system load of a clustered computing system (see figure 9). For each cluster node, a basic probe is necessary to observe the CPU load and the memory usage. Other system resources could be added to this use case: network bandwidth, disk transfer rate, etc. Now, getting all the measures from all these basic probes, as well as managing all these probes, is quite cumbersome. Conversely, composite probes enable getting a global system load indicator for the cluster obtained through a single probe that transparently handles control operations for the underlying sub-probes. Then, the global system load probe may be based on individual system load probes that aggregate measures from basic probes (CPU, memory). Finally, component/probe sharing enables an arbitrary number of different aggregations, such as the global cluster CPU load indicator provided by the cluster_CPU composite probe.

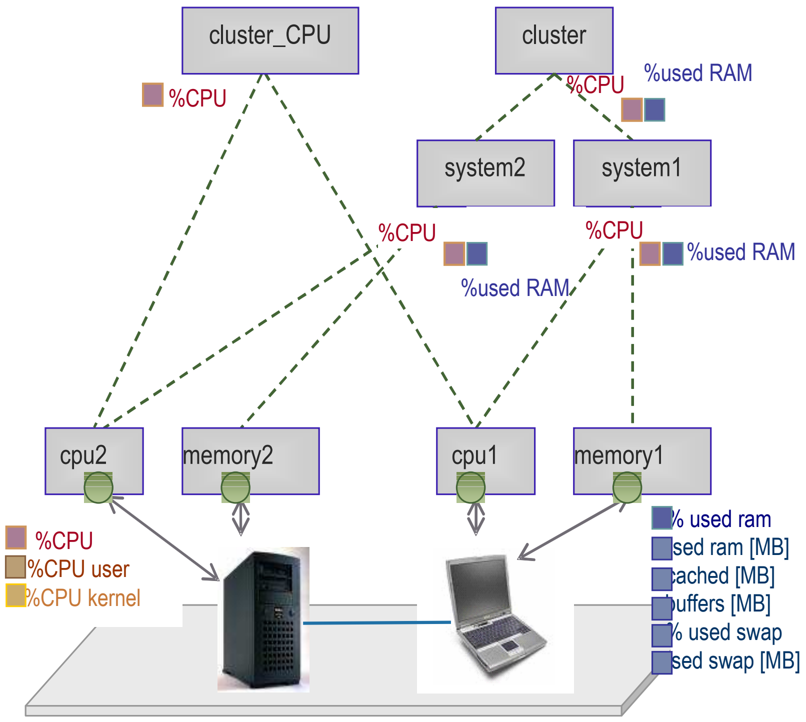


figure 9. Sample composite probes hierarchy for monitoring a clustered system

4.5.4 Integration to the Selfware platform

First, these probes are Fractal components, which is key to an easy integration to the Selfware platform. They will straightforwardly benefit from the deployment service. Then, they shall be slightly extended to become Managed Elements, in order to benefit from the Selfware autonomic features, such as self-repair (replace a faulty probe) or self-configuration (e.g. increase or decrease the sampling rate or measurements).

The link between Autonomic Managers and probes can be either direct, through plain Fractal RMI bindings, or indirect through some communication element dedicated to the transport of measures and commands. The main reason for using an intermediate element is both for fault tolerance and scalability issues:

- a node that is becoming unreachable must not result in a global freeze of every communication between AMs and probes;
- a large-scale system results in a huge number of probes that requires a special communication hierarchy (that can be addressed also by the composite probe architecture presented above).

This element could be introduced through a plain communication component, or through specific Fractal bindings. This point will be refined in the second version of Selfware's architecture, once we get feedback from experiences with the first version.

4.6 Checkpointing Managed Elements

4.6.1 Objectives

The main objective of the SR (System Representation) Service is to maintain a causally connected representation of the overall Selfware management system, including both its autonomic managers and its managed elements. The term *causal connection* means that relevant changes occurring in the management system are mirrored in the system representation and, conversely, that any changes in the system representation are reflected in the actual evolution of the management system.

The overall Selfware management system is built with Fractal components. In the case of the management of legacy software, the managed elements are components wrapping the management interface of the legacy into a uniform management interface defined by Selfware, as described in the SP1-L1 document. The System Representation is also composed of Fractal components, that exhibit a configuration state isomorphic with the configuration state of the actual management system. The notion of *configuration state* more precisely includes the components name, attributes, bindings, compositions, and lifecycle states.

All these aspects are isomorphic in the component-graph provided by the System Representation and in the component-graph corresponding to the Selfware management system, resulting in a dual architecture as illustrated by the following Figure.

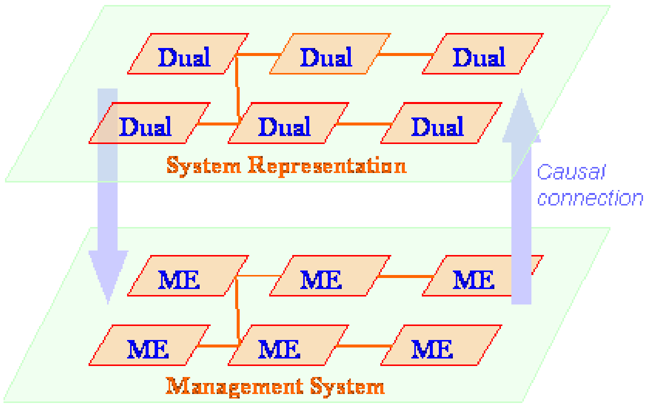
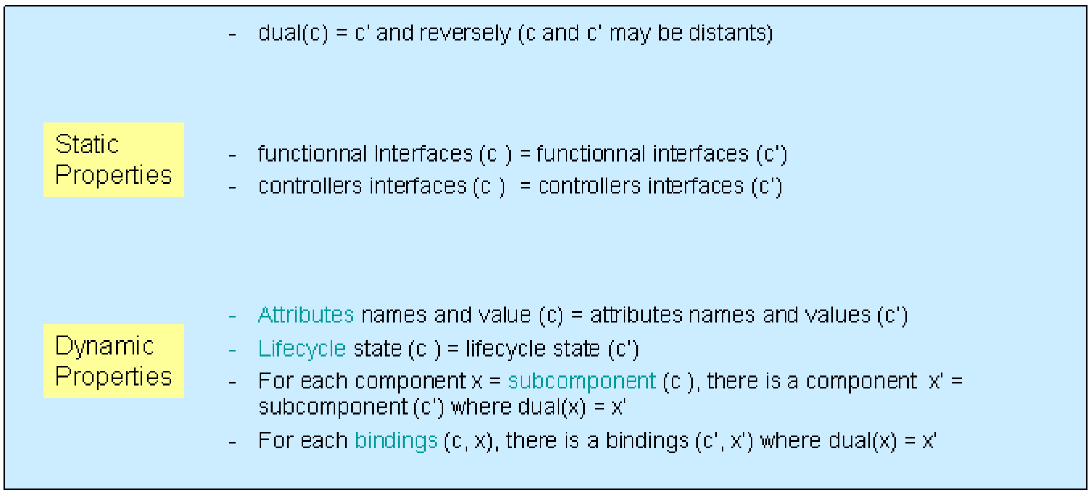


figure 10. Duality between the Management Layer and the System Representation

The causal connection between the Selfware management system and the SR (System Representation) is managed on a per-component basis. Any SR-component (meaning a component belonging to the System Representation) is causally connected to a component belonging to the Selfware management system and reversely. We say that such components are duals of each other.

Being given a component c belonging to the Selfware management system, and its dual in the System Representation c', the following properties are ensured by the SR service.



- dual(c) = c' and reversely (c and c' may be distants)

Static Properties

- functionnal Interfaces (c) = functionnal interfaces (c')
- controllers interfaces (c) = controllers interfaces (c')

Dynamic Properties

- Attributes names and value (c) = attributes names and values (c')
- Lifecycle state (c) = lifecycle state (c')
- For each component x = subcomponent (c), there is a component x' = subcomponent (c') where dual(x) = x'
- For each bindings (c, x), there is a bindings (c', x') where dual(x) = x'

The causal connection is maintained as follows:

- On bootstrap of the Selfware management system, the System Representation contains an architecture description of the system after initialization (including nodes that run the management system with the components they contain).
- Any update of the configuration state of the Selfware management system (e.g., adding a new node, deploying a new software element on a given node) is reported at the level of the System representation, and reversely.

4.6.2 Design Principles

As said in [3], a Fractal component is a runtime entity that has one or more business interfaces, as well as a predefined and extensible set of control interfaces. The control interfaces provide a meta-level access to a component internals allowing for component introspection and intercession. These interfaces are implemented by a set of extensible controllers. The main predefined Fractal controllers are those dealing with the configuration state of a component:

- The Name controller allows setting and getting the Fractal name of a component.
- The Attribute controller allows setting and getting the exported attributes of a component.
- The Binding controller mainly allows binding and unbinding the client interfaces of a component.
- The Content controller mainly allows adding and removing sub-components in a composite component.
- The Life-cycle controller mainly allows starting and stopping the threads of execution in a component.

The main design principles of the SR service are (1) to define an additional controller in charge of managing the bijection between a component and its dual, and (2) to modify the existing controllers in order to ensure that any configuration action performed on a component is reported on its dual. Following these design principles, the business part of a Fractal component stays untouched by the SR service that only deals with the control part (interface and implementation) of a component.

The modification of the pre-defined controllers is performed at two levels. First, the interfaces provided by the controllers have been extended in order to support a notification protocol. For instance, when a configuration action such as bind(..) is invoked on a component c, a corresponding action bindNotify(..) is invoked on c's dual component.

Second, the implementation of the controllers interfaces have been specialized in order to manage the causal connection with the dual components. More precisely, each time a *configuration action* is performed on a component, the controller performing this action has to perform an additional step that consist in notifying its “dual” controller. These design principles are illustrated in figure 11 and in figure 13.

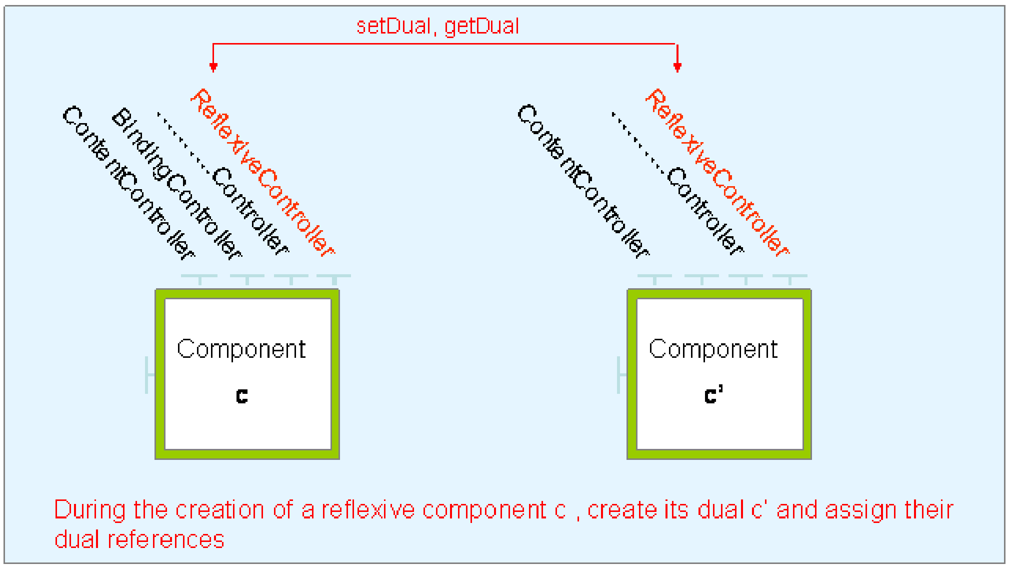


figure 11. Controllers associated to the components managed by the SR service

The specialization of the controllers implementation relies on the mixins technique used for implementing Fractal controllers in an efficient and extensible way [14]. This technique allows adding actions to be executed on behalf of a given control interface, in a very easy and incremental manner.

4.6.3 Component creation

In order to create components associated with a dual representation, a specific Fractal factory component is provided by the SR service. This factory is called “Reflexive Bootstrap Component”, and is itself a Fractal component associated with a dual representation that is called *dual factory*. Any creation of a component c performed by such a factory is notified at the dual factory, that consequently creates the dual component of c.

```
ReflexiveBootstrapComponent.newFcInstance(..):
        c = factory.newInstance();
        dualFactory = factory.getDual();
        dualc = dualFactory.newFcInstanceNotification(..);
        c.setDual(dualc);
        dualc.setDual(c);
```

figure 12. Actions performed at component creation time

More precisely, the reflexive bootstrap component performs the actions listed above when it is asked for a component creation. It starts by creating a first component instance, and then it asks its associated dual factory to create the component’s dual instance. Finally, it updates the dual references of the created components.

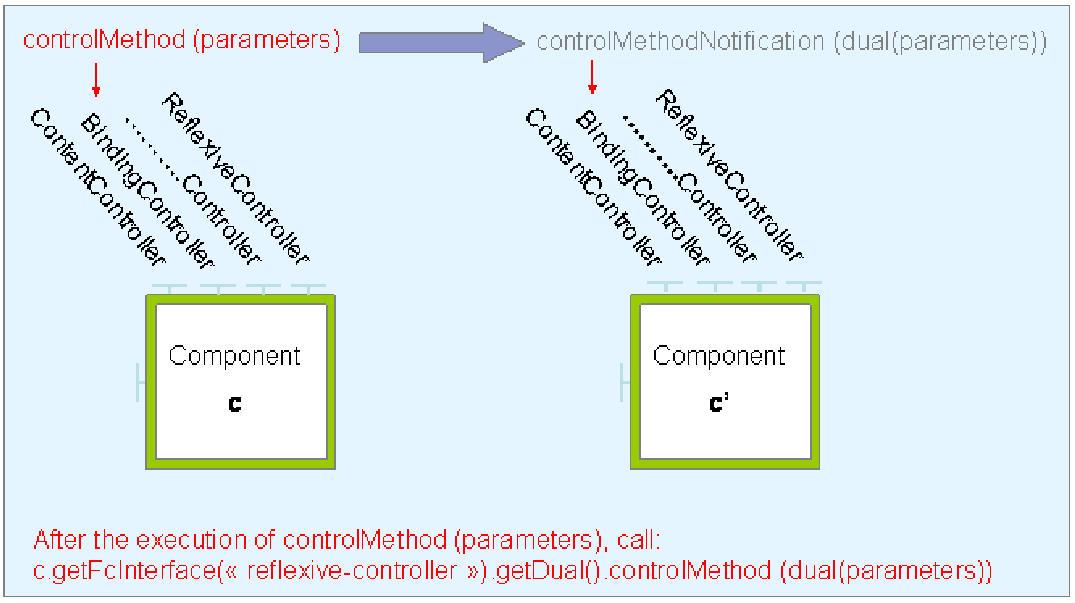

controlMethod (parameters)
controlMethodNotification (dual(parameters))
ContentController
BindingController
Controller
ReflexiveController
Component
c
ContentController
BindingController
Controller
ReflexiveController
Component
c'
After the execution of controlMethod (parameters), call:
c.getFcInterface(« reflexive-controller »).getDual().controlMethod (dual(parameters))

figure 13. Execution of a control method on a component managed by the System representation

5 Decision support for autonomic managers

5.1 Introduction

In the Selfware architecture, the Autonomic Manager is in charge of reacting when a defect is detected in the Managed Elements it is responsible for. It actually implements the logic the self-* features. The autonomic control loop starts with the monitoring function described in section 4.5, and ends by acting on the Fractal system through reconfiguration actions that would be typically supported as shown in section 4.4. In between, the decision making function may be integrated in a variety of ways (see discussion in section 4.5.4), and may be based on a variety of decision subsystems, ranging from basic algorithms hard-coded in a Java component, to a variety of artificial intelligence techniques: rule engines, Bayesian networks, constraints solving, neural networks or genetic algorithms.

Some existing works address various approaches for implementing decision module. Each solution is effective for solving specific types of problems. Traditional solutions based on heuristics [11],[16] aims to provides fast solutions for specific problems. Those fast solutions provide good result but have a limited scope. Extends a heuristic is not obvious and could dramatically decrease 1/ its performance 2/its quality.

By the usage of policies or rules, it is possible to implement more flexible decision modules. Such systems extends Policies Base Management Systems (PBMS) or Expert Systems. In the context of Autonomic Computing, such approaches use Event-Condition-Action (ECA) solutions [1]. In this situation, the decision module is composed by a set of rules. Each rule describes an action to execute when certain events occurred on certain conditions. Those approaches are more declarative than heuristics-bases solutions as they often use a specific domain language to writes the rules. However, those approaches have some limitations in certain environments. Some conflicts between rules could occur when there is a certain overlapping in the source, or the target. Those limitations could be minimized statically by providing some mechanisms in the languages or dynamically by observing results after the execution of a rule. The inclusion of meta-heuristics or priority between rules is another solution.

Last, approaches using Constraints Programming [trends, handbook] (CP) propose a more safe and flexible approach. A constraint solver aims to find solutions to a problem that satisfy a set of constraints. Those constraints are written by users. CP, by using propagation and pruning aims to solve better solutions. However, considering the size of the search-tree, these solutions could not be as fast as heuristic-based approach or PBMS.

Our goal is not to make a prescription on whether one technique is better than another one. Although giving a classification of the best approach with regard to categories of autonomic features and systems would be of major interest, we think that it is definitely out of reach for the Selfware project, and probably for today's autonomic computing community. Our pragmatic goal consists in beginning by trying a couple of decision-making techniques and to see how it can be integrated in an AM of our architecture. This section will be extended in a second version, featuring in more details other ways of supporting decision making.

5.2 Decision making with active rules (ECA rules)

Selfware deliverable SP1-L1 introduces the rationale and the main constituents of an active rule service that allows for the use of active rules as a basic decision making mechanism in component-based autonomic systems. SP1-L1 mentions major points that deserve a special attention: the design of the rule model (both rule definition model and rule execution model) and its architectural integration.

This section of SP1-L2 gives a more detailed view of the rule model and its architectural integration in the form of an active rule service.

5.2.1 Overview

Reactive behaviour, the ability to (r)eact automatically to take corrective actions in response to the occurrence of situations of interest (events) is a key feature in autonomic computing. In active database systems, this behaviour is typically incorporated by Event-Condition-Action (ECA or active) rules (when a specified event occurs, evaluate a condition predicate, and if true execute some action operations).

The approach we propose consists in defining a mechanism for the integration of these rules in component-based systems to augment them with autonomic properties. The contribution is twofold.

- First, the proposal of a rule model, i.e. a rule definition model together with a rule execution model, that can be coherently integrated into a component model.
- Second, the proposal of 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/implemented 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 implementation is extensible: additional components can be added at will to the library to render more elaborate and more specific semantics according to certain applicative requirements.

5.2.2 Rule Definition Model

The rule definition model specifies what are the form of events, conditions and actions considered and manipulated by the active rule service.

Event

An event is a happening of interest at a given point in time. It is characterized by an event type, i.e., an expression describing a class of significant occurrences of interest. The active rule service considers the following event types: i) *applicative* which corresponds to inter-component interactions, ii) *structural* which represents modifications (reconfigurations) of the structure (topology) of the system, like adding or removing a component, or destroying or creating bindings between components, iii) *system-level* which characterises events coming from the external or underlying environment or context of execution (e.g. JVM and OS events).

In Selfware architecture, events (probes) are detected and notified by the monitoring service to the rule service.

Condition

Conditions are optional and express additional constraints on the state of the system that must be satisfied for the action part to be executed. Condition expressions (e.g. whether an attribute's value is bigger than a particular value or not) are boolean expressions built using logical operators.

More complex expressions can be formed based on queries on the structure of the (component-based) system as well as on its behaviour. A query that selects the various components linked to a particular one is one such example. In case of a complex query that does not return a boolean result, the condition is considered to be true if the query returns a non-empty result.

In Selfware architecture, such conditions are expressed as FPath expressions and evaluated by the FPath engine which is part of the Selfware navigation service.

Action

The corrective (re)actions that the target system can be subjected to are expressed in the action part of the rule. The event and condition parts of the rule serve to analyse the symptoms affecting the system.

To rectify the anomalies, the action can range from simple parameterizations of component attributes, for example, an increase in the size of a cache or pool, to complex structural reconfiguration operations, which can include addition removal or replacement of one or several components. Other types of actions can be envisaged, like external notifications, for example, an email or SMS notification to an administrator. In Selfware architecture complex reconfiguration action plans are expressed as FScript expressions and executed by the FScript (transactional) engine which is part of the Selfware navigation service.

5.2.3 Rule Execution Model

Rule Execution Cycle

The entire execution of a single rule is comprised of the following phases and various states:

- Triggering and Event Processing Phase {R(E)}: this phase begins with the notification of the event(s) that triggers ("wakes up") the rule. The notification is performed by the entity on which the event occurs. It consists in processing the event(s) based on the various rule execution parameters. The rule goes from the *triggerable* state to the *triggered* state.
- Condition Evaluation Phase {R(C)}: the second phase of the execution evaluates the condition expression. If the condition is satisfied then the rule transits from the *evaluable* state to *evaluated* state.
- Action Execution Phase {R(A)}: the last phase of the rule execution corresponds to the execution of the action part of the rule. It takes the rule from the *executable* state to the final state of *executed* state (generally confounded with the initial *triggerable* state), thus inducing a positive feedback change in the system behaviour.

Execution units and execution points in component-based execution models

If active rules have an execution model (behaviour) of their own (cf. previous and following paragraphs), the introduction of active rules in a system (be it a database or a component-based system) has a non negligible impact on the behaviour of that system. Indeed, there exists a dependency between the execution of the system and the execution of rules, for it is the former that triggers the latter and also the two executions are interwoven/intertwined together.

A key difference between active database systems, where active rules have been extensively studied, and active component-based systems, where we want to apply active rules, is that execution models in active database systems are based on the central concept, that of *transaction*, which is (generally) inexistent in component-based systems.

Transaction is a core and foundational concept of active database systems because, thanks to transaction *demarcations* (*start*, *commit*, *abort/rollback*), they provide a natural and convenient *execution unit*} for the execution of active rules. An execution unit specifies an interval (between two execution points in a sequential flow or basically between two points in time) during which events can be detected/notified to interested rules and rules can be evaluated and executed. On the one hand, component-based systems generally do not consider transactions. On the other hand, the behaviour of a component-based system generally refers to interaction through interfaces only, thanks to operation invocation. Hence, we define the execution unit in component-based systems as delimited by the interval between two execution points: the reception of an operation invocation on a server interface and the emission of a response onto a client interface.

For method invocations on a component's functional interfaces (which produce applicative events), and operations that modify the structure of the system (which produce structural events), we may signal two events: *begin* and *end*. Other forms of events (e.g. system events) can be integrated in the model by considering that their begin and end events are merged (i.e. they both represent the same execution point or point in time).

Rule Execution Dimensions

Based on these definitions of execution units and execution points, adapted from those in active database systems, the approach for defining a rule execution model for autonomic component-based systems consists in defining a set of *dimensions* with their possible values; together with a specific (deterministic) behaviour/semantics to all combinations of dimensions and values. Indeed, the condition of a triggered rule is not always evaluated immediately (hence the two separate states *triggered* and *evaluable*; and a triggered rule with a satisfied condition is not always executed immediately (hence the two separate states *evaluated* and *executable*.

When and how (e.g. immediately or later, in then same activity/thread that the operation that generated the event or a new one, etc.) a rule is processed depends on the various *dimensions* of the rule execution model (behaviour model). Of course, when multiple rules are concerned, which is the case in real autonomic systems, an execution model also specifies when and how rules triggered simultaneously (by same or different events). This is handled by *rule execution strategies* or *policies* which basically specify the scheduling of rules (e.g. depth-first order, width-first order, flat order, by cycles in sequential or parallel settings).

We do not detail these aspects here (interested readers may refer to N. Jayaprakash PhD thesis). Roughly, the dimensions and values currently handled by the active rule service are the following: *execution mode*: [immediate, delayed]; activity mode: [same, separate]; event processing mode: [instance, set]; local execution strategy: [mixed sequential order and parallel]; Global execution strategy: [flat, depth-first, width-first].

5.2.4 Architectural Integration

The architecture of an autonomic infrastructure based on active rules is inspired from the fundamental management notion of *domain* [13] which consists in grouping the components on which the various reactive operations can be carried out. A domain is:

- *a unit of composition* that enables logical structural partitioning of components in a system, and
- *a unit of control* that defines the type of (reflexive) control performed onto these components.

The similarities between a Fractal component and the concept of domain suggest that a domain can be aptly modelled as a Fractal component. To incorporate reactive behaviour, several types of domains have been defined, each with a particular type of control unit applied onto its composition unit. They are each represented by a Fractal component, known as *active domain*. The autonomic infrastructure is formed by an assembly of such reactive domains - superimposed on the target system.

Architectural Design

The set of domains that form our autonomic infrastructure are listed below and their relationships illustrated by figure 13.

- An *Event (E)* domain contains software entities where events of interest need to be detected (generally applicative components). Its control unit is responsible for identifying the software entities (components) that would belong to the content, instrumenting them appropriately and processing the events on their occurrences.
- A *Condition (C)* domain contains software entities that represent the scope of the queries that are to be evaluated. The functions of its control unit includes identifying the components that would be in its jurisdiction, and evaluating queries on them.
- A *Action (A)* domain encapsulates software entities on which actions are to be executed. The type of control enforced involves identifying the constituents of the domain's content and executing the corrective operations on request.

- A Rule (R) domain comprises of exactly one instance of the above 3 subdomains, i.e., Event, Condition (optional) and Action. R's control unit is responsible for coordinating the execution of its subdomains.
- The content part of *Policy (P)*domain is composed of Rule domains, and its control unit possesses the rights to their execution.
- The entire control space is a single *Policy Manager (PM)* domain that contains all the Policy domains with the sole functionality of coordinating the execution of individual rules.

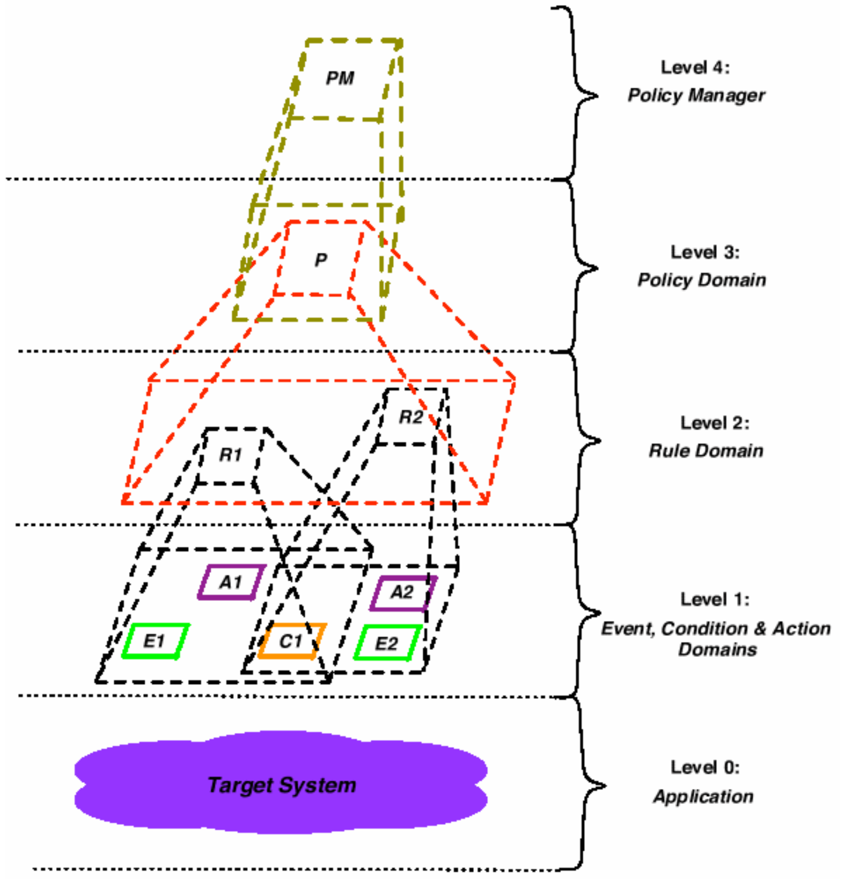


figure 14. Hierarchical Domains

The detailed design and implementation of active domains cannot be detailed here (cf. N. Jayaprakash PhD thesis work, [8]). It involved especially of careful design of components membranes of active domains where some of them are 'classical' controllers (plain Java) when others are themselves implemented as Fractal components where fine-grained reconfigurations are needed. Active domains together with their sub-domains and components and controllers used in their membranes form a modular and extensible framework/toolkit for the construction of ECA rule-based autonomic architectures which is part of the Selfware decision service.

6 References

[1] K. Appleby and S. Fakhouri and L. Fong and G. Goldszmidt and M. Kalantar and S. Krishnakumar and D.P. Pazel and J. Pershing and B. Rochwerger, Oceano-SLA based management of a computing utility, Proceedings of the Integrated Network Management International Symposium, 2001

[2] Sara Bouchenak, Fabienne Boyer, Noel De Palma, Daniel Hagimont, Sylvain Sicard, and Christophe Taton, JADE: A Framework for Autonomic Management of Legacy Systems, 2006

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

[4] CECCHET (E.) et al, Implementing Probes for J2EE Cluster Monitoring. Studia Informatica, 4(1), 2005.

[5] P.C.David, T. Ledoux, Safe Dynamic Reconfigurations of Fractal Architectures with FScript, Proceedings of the 5th Fractal Workshop at ECOOP, Nantes, France, 2006

[6] Ada Diaconescu, Automatic Performance Optimisation of Component-Based Enterprise Systems via Redundancy, in Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, California, USA, November 7-11, 2005

[7] Dillenseger (B.), Flexible, easy and powerful load injection with CLIF version 1.1. Fifth Annual ObjectWeb Conference, Paris La Défense, January 2006.

[8] Jayaprakash, T. Coupaye, C. Collet (LIG), P.C. David (INRIA). Flexible Reactive Capa-. bilities in Component-Based Autonomic Systems, International Conference on Autonomic Computing and Communication Systems (Autonomics 2007)

[9] M. Marjan, H. Jan, S. Anthony, When and how to develop domain-specific languages, ACM Computing Surveys, 2005

[10] OSGI Alliance. The OSGI Service Platform – Dynamic Services for networked devices. http://www.osgi.org

[11] Ruth, P. and Rhee, Junghwan and Xu, Dongyan and Kennell, R. and Goasguen, S., Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure, ICAC '06. IEEE International Conference on Autonomic Computing

[12] Selfware Architecture, Livrable SP1-Lot1, August 2007

[13] Sloman M, Twidle K., Domains: A Framework for Structuring Management Policy, Chapter 16 of Nerwork and Distributed System Management, Addison Wesley, 433-453.

[14] Gary T. Leavens, Murali Sitaraman, Foundations of Component Based Systems, Cambridge University Press, 2000

[15] World Wide Web Consortium, XML Path Language (XPath) Version 1.0, W3C Recommendation, 1999

[16] Jing Xu; Sumalatha Adabala; Fortes, J.A.B, ICAC 2005..Second International Conference on Autonomic Computing,