SCA Platform Specifications - Version 2.0

Damien Fournier, Philippe Merle, Gaël Blondelle, Lionel Seinturier, Nicolas Dolet, Christophe Demarey, Vivien Quéma, Valerio Schiavoni, Samir Tata, Djamel Belaïd, et al.

SCA Platform Specifications - *Version 2.0*

Funded by

AGENCE NATIONALE DE LA RECHERCHE
ANR
http://www.agence-nationale-recherche.fr

Colophon

Date	May 24, 2011
Deliverable type	Specification
Version	2.0
Status	Final
Work Package	1
Access permissions	Public

Editors

INRIA	Damien Fournier Philippe Merle

Authors

EBM WebSourcing	Gaël Blondelle
INRIA Adam team	Philippe Merle Lionel Seinturier Damien Fournier Nicolas Dolet Christophe Demarey
INRIA Sardes team	Vivien Quema Valerio Schiavoni
INT	Samir Tata Djamel Belaid
IRIT	Daniel Hagimont

Table of Contents

List of Figures

List of Tables

1 Introduction

The SCOrWare project aims at building an open source implementation of the Service Component Architecture (SCA) specifications. This implementation is composed of 1) a runtime platform for deploying, executing, and managing SCA-based applications, 2) a set of development tools for modeling, designing, and implementing SCA-based applications, and 3) a set of demonstrators. This document contains the specifications of the SCOrWare runtime platform. Section 1.1 lists the parts of SCA specifications supported by the SCOrWare platform. Section 1.2 gives an overview of the SCOrWare platform, and a summary of next chapters of this document. Section 1.3 lists the main update from the version 1 to the version 2 of this document

1.1 SCA specification coverage

The Service Component Architecture (SCA) specification, established by the Open SOA Collaboration [34], provides a programming model for building applications based on Service Oriented Architecture (SOA). The SCA specification mainly defines a model for the assembly of coarse grained and loosely coupled services and fine grained and tightly coupled components. On one hand, fine-grained/tightly-coupled components allow to define and compose particular business functions while on the other hand coarse-grained/loosely-coupled composite services permit to compose units deployed within different domains (e.g., different companies). Moreover, components in SCA assemblies can be implemented using several programming languages and composites can be linked using various binding and transport protocols. As a whole, the SCA 1.0 specifications are composed of 15 documents.

SCA Specification Document	Covered	Date	Comment
Assembly Model V1.00	Yes	21/03/07	Addressed in Chapters 2, 3, and 8.
Policy Framework V1.00	No	21/03/07	Out of the initial SCOrWare work plan.
Transaction Policy V1.00	Yes	03/12/07	Addressed in Chapter 6.
Java Common Annotations and APIs V1.00	Yes	21/03/07	Addressed in Chapter 4.
Java Component Implementation V1.00	Yes	21/03/07	Addressed in Chapter 4.
Spring Component Implementation V1.00	Yes	21/03/07	Not described in this document but supported by the OW2 FraSCAti open source software.
BPEL Client and Implementation V1.00	No	21/03/07	Out of the initial SCOrWare work plan.
C++ Client and Implementation V1.00	No	21/03/07	Out of the initial SCOrWare work plan.
Web Services Binding V1.00	Yes	21/03/07	Addressed in Chapter 5.
JMS Binding V1.00	No	21/03/07	Out of the initial SCOrWare work plan.
EJB Session Bean Binding V1.00	No	21/03/07	Out of the initial SCOrWare work plan.
COBOL Client and Implementation V1.00	No	24/07/07	Out of the initial SCOrWare work plan.
C Client and Implementation V1.00	No	24/07/07	Out of the initial SCOrWare work plan.
JCA Binding V1.00	No	05/11/07	Out of the initial SCOrWare work plan.
Java EE Integration Specification V1.00	No	03/07/08	Out of the initial SCOrWare work plan.

Table 1: SCOrWare specification coverage

While the current SCA specifications cover numerous languages for component implementation and various binding protocols, we choose, in the context of the SCOrWare project, to focus on providing an SCA platform for components implemented with the Java programming language. This means that SCOrWare covers specifications (Table 1) of (i) the assembly model, (ii) the Java Common Annotations and APIs and (iii) the Java component implementation. Rather than providing an alternative platform for supporting all the SCA specifications, the SCOrWare main objective is to address extra capabilities (i.e. Fractal and JBI integration, dynamic binding and autonomic deployment). Better support for

different component implementation languages is already provided by other SCA platforms like Apache Tuscany SCA [2]. However, the SCOrWare platform must be as flexible as possible to allow later support for other component implementation languages and binding protocols.

1.2 Overview of the SCOrWare Platform

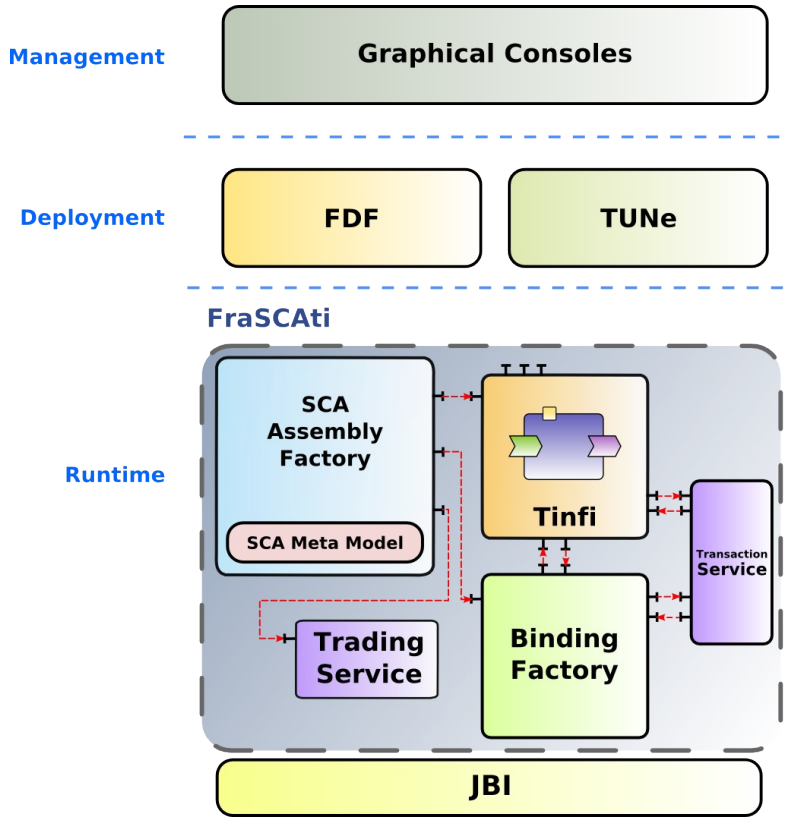


Figure 1: The SCOrWare runtime platform

As shown in Figure 1, the SCOrWare runtime platform, named FraSCAti for "Fractal-based SCA Implementation"[1], is composed of the following main parts:

- **The SCA metamodel**: This metamodel encompasses all the concepts defined in the SCA specifications. This metamodel is used by the FraSCAti assembly factory to parse SCA descriptors in order to reify them as models in memory. Let's note that the same metamodel is also used by SCOrWare Eclipse-based tools in order to design SCA-based applications and generate SCA descriptors automatically. This metamodel was obtained by processing the SCA XSD files via the Eclipse Modeling Framework (EMF). Before that, the SCA XSD files required to be corrected for removing inconsistencies. The SCA metamodel is described in Chapter 3.
- **The Tinfi runtime kernel**: Tinfi, stands for "Tinfi Is Not a Fractal Implementation", is the implementation of the FraSCAti runtime for Java-based SCA components. Tinfi is built as an extension of the Fractal component model and its Java-based reference implementation, i.e., the Julia framework. With Tinfi, Java-based SCA components are simultaneously both SCA-compliant and Fractal-compliant. The main benefits is to make all SCA components dynamically reconfigurable at runtime. Let's note that this extra-functional property is not addressed by SCA specifications and is not supported by concurrent SCA implementations like Apache Tuscany. Tinfi is described in Chapter 4 Section 1.
- **Integration with JBI** - The SCOrWare runtime platform can be packaged as a JBI component in order to be

1 Frascati is also an Italian town when Tuscany is an Italian region

deployed on top of the open source OW2 PEtALS platform, a distributed JBI bus. A new SCA binding for JBI is also specified to allow SCA components to access to or to be accessible from other JBI components. The integration with JBI is described in Chapter 4 Section 2.

- **The Binding Factory** - The binding factory is in charge to establish communication paths between SCA components. This encompasses SCA bindings associated to both SCA composite services and references, promotion of both SCA component services and references, and local SCA wires between components. This binding factory supports various networked binding protocols like SOAP or RMI, and is built on top of the Fractal/Julia framework. The binding factory is described in Chapter 5.
- **The Transaction Service** - The SCOrWare transaction service allows to support transactional properties (Atomicity, Coherency, Isolation, Durability) when composing distributed SCA components. The transaction service is described in Chapter 6.
- **The Semantic Trading Service** allows the SCA applications to discover dynamically components (published in a registry) that satisfy a semantic description. Such a description associates to each component element (service, reference, property, implementation, ..) a semantic concept in a semantic model. Thus, the trading service allows application based on SCA components to be able to replace unavailable component at run time with equivalent ones. When the discovered component is functionally equivalent to the required one but not exactly adapted to the needed interaction mechanisms the trading service can use the binding service to solve the incompatibility. The semantic trading service is described in Chapter 7.
- **The FraSCAti Assembly Factory** – The FraSCAti assembly factory is in charge to parse SCA descriptions, validate them, and instantiate SCA components. The parsing step uses the SCA metamodel to reify SCA descriptors as models in memory. The validation step visits memory models and checks the constraints defined in the SCA Assembly Model specification. Finally, the instantiation step invokes Tinfi and the binding factory to translate SCA models into executable components and bindings. The FraSCAti assembly factory is described in Chapter 8 Section 1.
- **SCA System Deployment with FDF** – Deploying[2] an SCA system requires to deploy SCA-based applications but also SCA runtime environments like FraSCAti or Apache Tuscany and their required underlying software like Java Runtime Environments, Tomcat servlet engines, and PEtALS JBI containers. The Fractal Deployment Framework (FDF) allows system administrators to describe and deploy any complex stack of software on top of distributed physical infrastructure. In the context of SCOrWare, new FDF personalities were defined to deploy SCA applications, and both FraSCAti and Apache Tuscany runtime environments. The deployment of SCA systems with FDF is described in Chapter 8 Section 2.
- **Autonomic SCA applications with TUNe** - TUNe will be used to support autonomous SCA systems. Autonomic support with TUNe is described in Chapter 8 Section 3.
- **Graphical management consoles** - Graphical management consoles are described in Chapter 9.

This document contains also three other chapters:

- Chapter 2 is about the unification of architecture-based and service-oriented approaches. Particularly, this chapter presents SCA, JBI and Fractal component models, and discusses our SCOrWare vision on how these models can be put together at work.
- Chapter 10 is about the integration of the various parts composing the FraSCAti runtime platform.
- Chapter 11 concludes this document.

1.3 Changes with Platform Specifications 1.0

Compared to the specification 1.0, part of the document has been revised in compliance with current implementation of the SCOrWare Platform. Changes with the latest document are listed below :

- Chapter 4, precisely section 4.1 has been partially updated with the latest tinfi kernel implementation,
- The binding factory described in chapter 5 has also been updated according to its current implementation; Part of the chapter describes API and extension mechanism of the binding factory.
- Chapter 6, which defines the transaction service, has been added to this document.
- Chapter 7 about semantic trading and composition has completely been replaced.
- The implementation description for the Assembly factory (Section 8.1.4) and the System deployment (Section 8.2.3) have been added to the chapter 8; And Autonomic Support (Section 8.3) has been replaced.
- Graphical Console described in Chapter 9 has been added.

2 Deploying stands for uploading, configuring, and activating software.

2 Unification of Architecture-based and Service-Oriented Approaches

This chapter contains the state of the art related to the Task 1.1 of the SCOrWare project. Section 2.1 presents our SCOrWare vision about the unification of architecture-based and service-oriented approaches. Section 2.2 discusses the complementaries between existing service-oriented and software component models. Particularly, we focus on SCA, JBI and Fractal models introduced in Section 2.3, 2.4, and 2.6, respectively. Finally, we discuss how these models can be put together at work. How mapping SCA to JBI concepts is discussed in Section 2.5 and how mapping SCA to Fractal concepts is discussed in Section 2.7.

2.1 Service-Oriented Vs Architecture-Based Approaches

In the domain of distributed software infrastructures and middleware, we have recently seen the emergence of the two visions of the organization of these infrastructures:

- **Architecture-based organization.** In this vision, a distributed software system is organized as an architecture built as an assembly of software components with well defined interfaces. The important concept underlying software architectures is that the architecture of an application is described at design time with an architecture description language (ADL), that gives a base to express treatments to be associated with this architecture, like its deployment in a target environment, or its specialization by associating with it the definition of non functional properties (transactions, security). In some architecture-based middleware, the architecture of the application may evolve dynamically (thus implementing reconfigurations) and its description is maintained during execution, which allows expression of behaviors that are applicable to evolving architectures. In an architecture-based organization, we generally have a (relatively) strong coupling between the assembled components, because we suppose that the interfaces of these components are well known and compatible to enable the interconnection. The ObjectWeb Fractal and OMG CORBA component models are two examples of middleware supporting architecture-based organizations.

1. **Service-oriented organization.** In this vision also called Service Oriented Architecture (SOA), a distributed software infrastructure is organized as a set of independent services. The services are arranged by suppliers and in general, they are accessible via Web services and XML documents (to describe the format of data exchanged between the services). The important characteristic in this approach is the light coupling between the services used in an application. The interfaces of a service are described as XML documents, which promotes the interoperability between the services coming from independent suppliers. These independent services are supposed to be managed in the context of a specific application by an orchestration process which defines and coordinates the interactions and exchanges between services. Many Enterprise Service Bus (ESB) technologies are currently emerging and provide support for service-oriented organizations (e.g. Java Business Integration - JBI).

These two approaches both have their own advantages. Architecture-based organizations provide support for the applications management thanks to the notion of software architecture. Service-oriented organizations enhance independence between services, the integration of the services being defined in the orchestration process. However, both approaches rely on the notions of software brick (component or service) and application description (architecture or orchestration).

In 2005, a new standard, SCA (Service Component Architecture), emerged and suggests a unification of these two approaches. The purpose is to benefit from the advantages of both organizations, i.e. the management of independent services via a Web service interface and the management of software architectures enabling adaptations.

SCA defines a model which allows describing an SCA application as an assembly of SCA components, i.e. the instances of SCA components to be created, the configuration of their business and technical properties, the bindings to establish between components, etc. Assemblies of SCA components are described in XML descriptors. These XML descriptors altogether form a description similar to an Architecture Description Language (ADL).

SCA defines two component programming models, one for the C++ language and the other for the Java language. These models define the programming rules to be respected as well as the programming interfaces to be used for the implementation of SCA components in C++ and Java. The Java model relies heavily on the concept of annotation introduced in Java 5. Annotations allow the introduction of SCA specific meta data in the component implementation source code (provides and required services, configurable properties, etc.). An SCA platform must use these annotations in order to automatically produce the XML deployment descriptors and the technical code to be injected in the component implementations. This approach has already been used in component based platforms such as EJB 3.0 and Fractal/Fraclet.

2.2 Beyond the war of service-oriented and component models

Even if service-oriented and component-based approaches are well accepted, there is a plethora of industrial and academic models like Unified Modeling Language 2 (UML2), CORBA Component Model (CCM), Service Component Architecture (SCA), Java Business Integration (JBI), Spring, Enterprise JavaBeans (EJB), OSGi, OpenCOM, Fractal, and so on. Each model defines its own set of concepts to build component-based applications. This encompasses concepts like component type and instance, configurable attribute/property, port/interface, binding/wire, primitive versus composite component, pure hierarchical model or with sharing, etc. Moreover, each model defines an (open or close) set of management operations to apply to their concepts: create component types and instances, configure attribute/property values, establish binding between component ports/interfaces, etc. According to the used model, these operations can be executed at a certain step of the software life cycle. Here, we distinguish four different steps: Design time, assembly time, deployment time, and runtime. For instance, creating a new component instance is a design time operation in UML2, an assembly time operation in SCA, a deployment time operation in JBI, and finally a runtime operation with Fractal. Obviously, certain models claim that they are the universal and ultimate model to apply in every software application domain and at every step of the software life cycle. But in reality, most of the models have their own application domains and are appropriate (or not) for certain steps of the software life cycle. Table 2 gives a list of well-known component models, and identifies, for each of them, the software life cycle steps where they can be used, and their main application domains.

Name	Used at	Main application domains
Unified Modeling Language 2 - UML2	Design time	Every application domain
CORBA Component Model - CCM	Assembly time to runtime	Distributed real-time and embedded systems
Service Component Architecture - SCA	Assembly time	SOA applications
Spring	Assembly time	JEE Web Applications
OSGi	Deployment time	Java-based applications
Java Business Integration - JBI	Deployment time	Java-based SOA platforms
OpenCOM	Runtime	Middleware systems
Fractal	Runtime	Every application domain

Table 2: Some component models

UML2 is a modeling language that allows the design of component-based applications for (potentially) any application domain. But UML2 only provides a design time component model that does not impose that applications are assembled, deployed, and executed as assemblies of runtime components. For instance, a UML2 model can be mapped to a pure Java-based object-oriented program.

CCM is a component model mainly used to assemble, deploy, and execute distributed real-time and embedded component-based applications. CCM is addressed in more details in the scope of the ANR Flex-eWare project[3].

SCA is a service-oriented and component-based model to build SOA applications as assemblies of components. SCA provides an XML-based language to assemble components and some programming models to program these components. However, SCA does not impose that the associate runtime is also component-based or component-oriented. Then at runtime, SCA applications can be composed of object instances. SCA is described in more details in Section 1.3.

Spring is a component model to build JEE Web applications. Spring provides an XML-based language to describe applications as assemblies of components (named Spring beans). However, at runtime, Spring applications are not necessary component-based and mainly composed of pure Java-based objects.

OSGi provides a component model to build any kind of Java-based service-oriented applications. At deployment time, OSGi users can reason in terms of assemblies of components thanks to a dedicated XML-based language. But at runtime, OSGi applications are composed of pure Java objects.

JBI is a component model to build SOA platforms which will host SOA applications. A JBI component is a deployment unit but at runtime, JBI components are Java objects. JBI is described in more details in Section 1.4.

3 https://srcdev.lip6.fr/trac/research/flex-eware/

OpenCOM is a component model dedicated to build reflective middleware like ORB. With OpenCOM, components, interfaces, bindings are runtime entities which can be manipulated through a well defined meta protocol.
Fractal is a general purpose component model which has been applied at all software application layers: From system-on-chips, operating systems, middleware, containers, application servers, and to applications. Fractal is described in more details in Section 1.6.

In the scope of the SCOrWare project, we will combine several of these component models. UML2 will be used to analyse and design SCOrWare demonstrators and SCA-based applications. SCA will be used as the component model to build SOA applications at assembly time. JBI will be used as the component model to deploy and host both SCA runtime and applications. Finally, Fractal will be used as the component model to implement the SCA and JBI runtime.

2.3 The SCA Component Model

SCA (Service Component Architecture) is a set of specifications for building applications and systems using the principles of Service Oriented Architectures (SOA). The model has been promoted by a group of companies, including BEA, IBM, IONA, Oracle, SAP, Sun and TIBCO. The specifications are now defined and hosted by the Open Service Oriented Architecture (OSOA) collaboration[4].
One of the purposes of SCA is to define a component model for service architectures. SOA such as the ones based on Web Services provides a way for exposing coarse grained and loosely coupled services which can be remotely accessed. Yet the SOA approach seldom addresses the issue of the way these services should be implemented. SCA wishes to fill this gap by providing a component model. SCA entities are software components which provide and require services. The model is hierarchical with components being implemented either by primitive language entities or by subcomponents. SCA is neutral with respect to programming languages and several mappings exist for Java, C++, BPEL or Spring (other mappings are currently being investigated for PHP, C and COBOL). SCA is also neutral with respect to communication protocols between remote components. The most widely used solution is certainly SOAP, but the specifications can accommodate many other forms of communication bindings such as JMS, RMI or local method calls for components collocated in the same address space. SCA comes with an XML-based architecture description language.
Version 1.0 of the SCA specifications has been released on March 15, 2007. Previously, a first draft had been published in November 2005 (version 0.9 of the specifications). Two other intermediate versions have been released: 0.95 in July 2006, and 0.96 in August 2006. One should note that many differences exist between these versions. Several additions, removals and modifications have been applied by the authors. One can hope that the newly released 1.0 version will be more stable and mature.
Several implementations of the SCA model are available. Most of them are commercial implementations: HydraSCA from Roque Wave Software, IBM WebSphere Application Server Feature Pack for SOA, or Oracle Event-Driven Architecture Suite. The OSOA web site provides a comprehensive list of available solutions.
In the open source world, the leading implementation is Tuscany[5] which is an Apache incubator project. Two versions of Tuscany exist: one for Java and one for C++. Two other facts are worth noticing in the open source world: Newton[6] provides another implementation which however looks less advanced than Tuscany; three of the Tuscany developers have decided to fork and to create the Fabric3[7] project which is thus the third known open-source implementation of SCA.
The SCA specifications come with a series of companion specifications. The main one is SDO (Service Data Objects) which provides a unified way of handling data within applications. Another specification worth mentioning is the SCA Policy Framework which aims at capturing non-functional requirements for components through the specification of constraints, capabilities and quality-of-service expectations. The programming language mappings and the communication protocol bindings constitute another set of specifications which complement SCA ones.
To conclude this short introduction to SCA and before proceeding to its detailed presentation, let's mention the fact that SCA and SOA are active topics of the moment either for companies or for standardization bodies. As a matter of example, the companies which are at the origin of the SCA specifications launched recently (April 2007) the Open Composite Services Architecture (Open CSA) initiative within OASIS, which is a consortium for e-business standards. Open CSA will promote the further development and adoption of the Service Component Architecture (SCA) and Service Data Objects (SDO) families of specifications.

4 http://www.osoa.org
5 http://incubator.apache.org/projects/tuscany.html
6 http://newton.codecauldron.org
7 http://fabric3.codehaus.org/

2.3.1 Programming Artifacts

The main documents which describe the SCA component model are:

- the Assembly Model Specification [14],
- the language mapping documents:
  - for the Java language: the Java Component Implementation Specification [19] describes the way SCA components must be implemented in Java, and the Java Common Annotations and APIs [18] which summarizes, as the name suggests it, the available programming interfaces,
  - for the C++ language: the Client and Implementation Model Specification for C++ [15],
  - for the BPEL language: the Client and Implementation Model Specification for WS-BPEL [16] (in this case, a BPEL process is used as the implementation of an SCA component),
  - for the Spring framework: the Spring Component Implementation Specification [21] (in this case, a Spring application is used as the implementation of an SCA component).
- the communication protocol binding documents:
  - for Web Services: the Web Service Binding Specification [22] describes the way SCA component interfaces can use the SOAP protocol to remotely communicate,
  - for JMS: the JMS Binding Specification [20] describes the way SCA component interfaces can use the JMS protocol to remotely communicate,
  - for EJB: the EJB Session Bean Binding Specification [17] describes the way SCA component interfaces can be mapped to EJB stateless session beans. This provides a communication bridge between the SCA and Java EE worlds where SCA components can be exposed as EJB and reciprocally.

The main artifact of the SCA model is the notion of a **component**. As illustrated on Figure 5, an SCA component is a software entity which provides **services** (incoming arrows on the left-side of the box) and requires other services (outgoing arrows on the right-side). Required services are designated under the term of **references**. An SCA component may also export **properties** which are configurable data values. As mentioned in the introduction of this section, several languages may be used to implement an SCA component. The two most common ones are Java and C++. The Tuscany platform also provides non-standardized mappings to languages such as JavaScript and Ruby. SCA component services and references define **bindings** to communication protocols: a binding defines the particular technology which is used for interacting with the component. Each service and reference can be bound to a different protocol: this opens the way for components remotely communicating with different technologies. All components are identified by a **component name**.

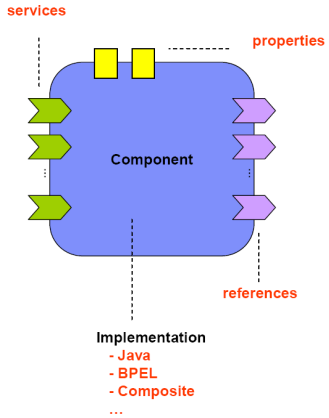


Figure 2: SCA component (from [14]).

An SCA component may be implemented with a programming language entity (e.g. a Java or C++ class) or may be implemented with an assembly of other SCA components. In this case, the component is said to be **composite**. Figure 2 illustrates this composition mechanism. Composites are also components with services, references and properties. Sub-components can be **wired** together. Service and reference types are defined with an **interface** definition language. As illustrated in the figure, two possible solutions are WSDL and Java. Services and references are also bound to a particular communication technology: SOAP over HTTP and message-oriented communications with JMS (Java Messaging Service) are the two most common solutions. In addition, as illustrated in the figure, other bindings have been defined: Java Connector Architecture (JCA) and EJB Stateless Session Bean (SLSB). Finally, services and references can be bound to a mechanism which is said to be SCA specific. In this case, the choice of the communication

technology is specific to the SCA platform which is being used for running the application. For example, Java platforms may use local object references and method invocations. Composites provide **composite services** and require **composite references**. These services and references are **promoted** from the subcomponents. As for regular components, **composite bindings** to communication technologies are defined for composite services and references.

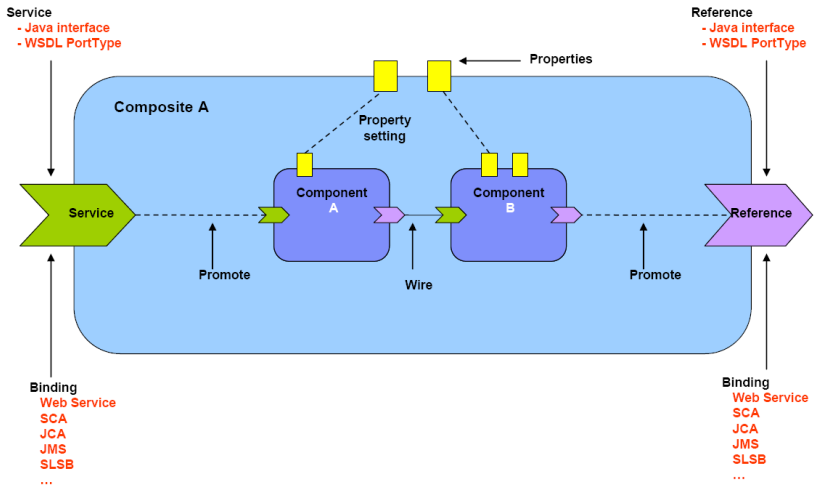


Figure 3: SCA composite (from [14]).

2.3.2 Java Component Implementation

The SCA specifications for Java define an API with 6 interfaces, 25 Java 5 annotations and 4 exception classes. These artifacts allow implementing in Java an SCA component defined in a SCDL assembly.
An SCA component is implemented by a Java class. Three main annotations exist for specifying provided services (`@Service`), required references (`@References`) and component property (`@Property`). Interested readers can refer to [19] for the detailed definition of the other 16 annotations.
In most cases, a service provided by an SCA component is described by a Java interface. Alternatively, a service may be described by a Java class (in this case all public methods of the class are considered to be provided services) or an interface generated from a WSDL portType. In all cases, the `@Service` annotation is associated to the Java class which implements the component and specifies which are the types of the services provided by this component.
A service required by an SCA component is materialized with the `@Reference` annotation in the implementation class.
The dependency towards the target component will be injected by the SCA platform based on the availability of this annotation[8]. Three modes of dependency injection are supported by SCA:

- field injection: `@Reference` annotates a public or protected field and the reference of the target component is injected directly in this field,
- setter injection: `@Reference` annotates a public or protected setter method and the value of the target component is injected by calling this method,
- constructor injection: `@Reference` annotates a parameter on a constructor and the value of the target component is injected when the component is instantiated.

In all three cases, Java reflection is put at work to retrieve the definition of annotated elements and to dynamically invoke them.
A property defined by an SCA component is defined with the `@Property` annotation which is associated either to a field or a setter method.

2.3.3 Example

We illustrate the previous notions with a simple example taken and adapted from the sample applications which are distributed with the Tuscany [2] implementation of the SCA specifications. The architecture of the example, named `SimpleBigBank`, is illustrated in Figure 6. This composite component manages bank accounts (checking, savings

8 Note that as a side effect, this requires the annotations to be retained at runtime in the `.class` file.

and stock accounts) for customers.

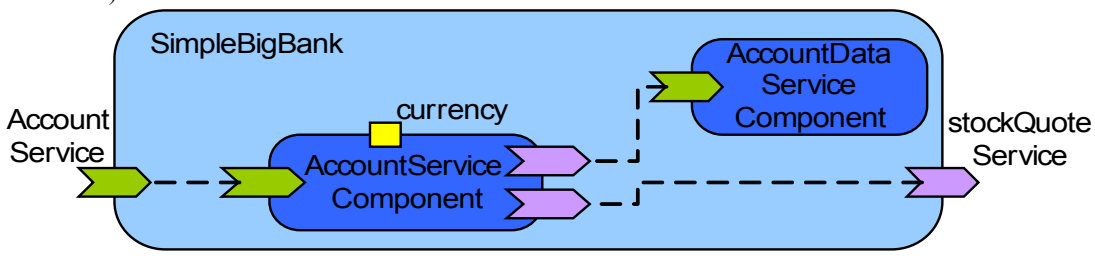


Figure 4: SimpleBigBank (from [10]).

The `SimpleBigBank` composite component contains two subcomponents: `AccountServiceComponent` and `AccountDataServiceComponent`. `SimpleBigBank` defines a service (`AccountService`) and requires a reference to a `stockQuoteService`. The service and the reference are promoted from the subcomponent `AccountServiceComponent`.
The following piece of code gives the definition of the `SimpleBigBank` composite with the syntax of the SCA Assembly Language.

```
01 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
02   targetNamespace="http://SimpleBigBank" name="SimpleBigBank">
03   <service name="AccountService" promote="AccountServiceComponent">
04     <interface.wsdl
05       interface="http://SimpleBigBank#wsdl.interface(SimpleBigBank)" />
06     <binding.ws uri="http://localhost:8085/AccountService" />
07   </service>
08   <component name="AccountServiceComponent">
09     <implementation.java class="bigbank.account.AccountServiceImpl" />
10     <reference name="accountDataService"
11       target="AccountDataServiceComponent" />
12     <property name="currency">USD</property>
13   </component>
14   <component name="AccountDataServiceComponent">
15     <implementation.java class="bigbank.accountdata.AccountDataServiceImpl"/>
16   </component>
17   <reference name="stockQuoteService"
18     promote="AccountServiceComponent/stockQuoteService"/>
19 </composite>
```

The composite component (`<composite>` tag on lines 01 and 02) is named `SimpleBigBank` (attribute `name` on line 02). The tag `<service>` (lines 04 to 07) defines the service `AccountService` which is promoted from (implemented by) the component `AccountServiceComponent` (attribute `promote` on line 03). The type of the `AccountService` service is defined by the `SimpleBigBank` WSDL file (lines 04 and 05) and is bound to a Web Service (line 06). The tag `<reference>` (lines 17 and 18) defines the reference `stockQuoteService` which is promoted from the `stockQuoteService` reference defined by the `AccountServiceComponent`. This reference should be provided by an external service.
The `AccountServiceComponent` (lines 08 to 13) is implemented in Java with the `AccountServiceImpl` class (line 09), requires a reference (`accountDataService` on line 10) which is wired to the `AccountDataServiceComponent` (line 11), and defines the `currency` property which is initialized with the `USD` value (line 12). Although not explicitly stated in lines 08 to 13, the promotion defines in lines 17 and 18 for the enclosing composite implies that the `AccountServiceComponent` also owns a reference to a `stockQuoteService`.
Finally the `AccountDataServiceComponent` (lines 14 to 16) is implemented by the `AccountDataServiceImpl` Java class.
Besides being composed with the Assembly Language, SCA components need to be implemented. As said before, several mappings to programming languages have been defined (including Java, C++ and BPEL). In the above example, components are implemented in Java. As a matter of example, the following piece of code illustrates the implementation of the `AccountServiceComponent`.

```
01 @Service(AccountService.class)
```

```
02 public class AccountServiceImpl implements AccountService {
03   @Reference public AccountDataService accountDataService;
04   @Reference public StockQuoteService stockQuoteService;
05   @Property public String currency;
06   public AccountReport getAccountReport(String s) { ... }
07 }
```

The component is implemented by the **`AccountServiceImpl`** class (line 02). The `@Service` annotation (line 01) specifies that this component defines a service which is implemented by the **`AccountService`** interface. Although not shown here, this interface defines a unique `getAccountReport` method which is implemented in line 06 (code details skipped for clarity sake). The component defines two references (`@Reference` annotated fields `accountDataService` and `stockQuoteService` on lines 03 and 04 respectively), and a property (`@Property` annotated field `currency` on line 05).

2.4 The JBI Component Model

This section presents the main concepts of the Java Business Integration specification (JSR 208 [32]), and describes more specifically the concept of "component" as introduced in this specification. First, we introduce the main goals of JBI, and how JBI is can be seen as a standard to build integration platforms. Then, we explain more extensively the communication between components through the JBI environment, as well as component component life cycles.

2.4.1 JBI, a standard for SOA

Java Business Integration (JSR 208 [11]) specification defines a standard mean to assemble integration components in order to create integration solutions to support SOA (Service Oriented Architecture) in an Enterprise Information System. Components are plugged into a JBI environment and can provide or consume services through the Normalized Message Router, in a loosely-coupled way. The JBI environment routes the exchanges between those components, and offers a set of technical services, specifically services for administration and life cycle management. These services provide the capability to build integration infrastructures that can evolve at runtime. Two kinds of components can be plugged on the NMR :

- Service Engines provide "integration logic" in the environment, such as XSL transformation, BPEL orchestration and so on.
- Binding components are sort of "connectors" to external services or applications. They allow communication with various protocols, such as SOAP, JMS, ebXML, etc.

JBI is built on top of state of the art SOA standards:

- service definitions are described in WSDL
- components exchanges XML messages following the document oriented model.

The JBI environment (also called JBI container) provides the glue between JBI components by acting as a Message Router to:

- Find a service provided by a component,
- Send request to a service-provider,
- Manage the message exchange life cycle (request, response, acknowledgements, etc.).

On the another hand, the JBI container manages, through a rich management API, the installation and the life cycle of the components, and the deployment of artefacts to configure an installed component (for instance, XSL stylesheets for a transformation service engine).

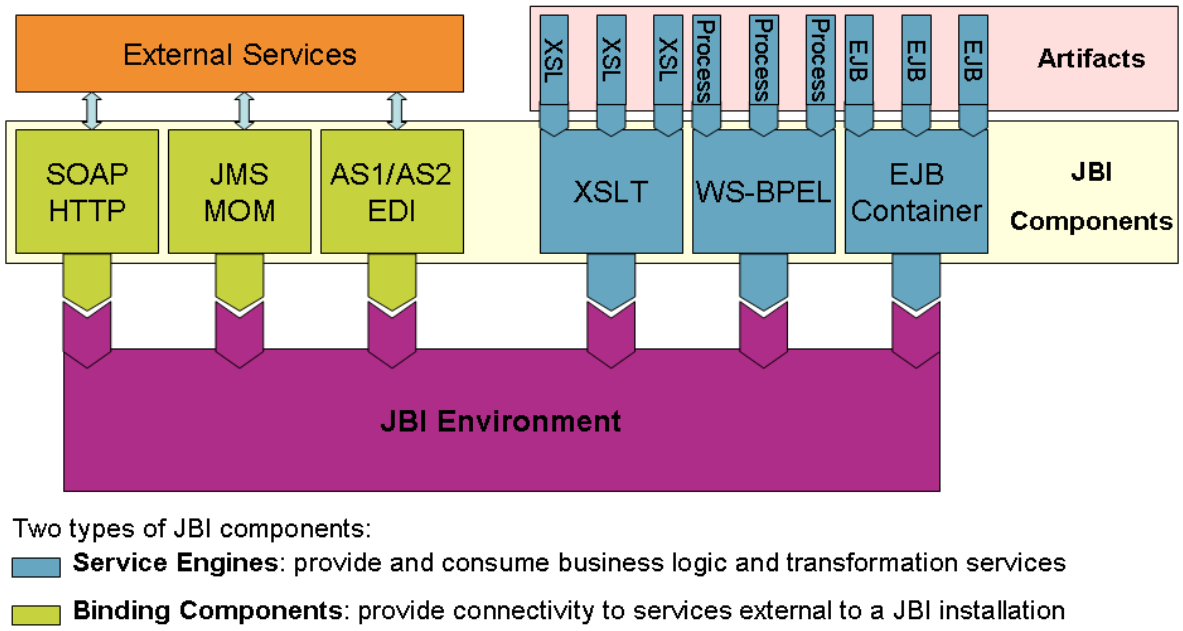


Figure 5: JBI container

2.4.2 JBI Components

JBI Components are the base elements to be composed by the JBI container in order to create an integration solution. The Components are plug-ins for the container, and are considered as "external". As a J2EE container hosts EJBs, or a Portal hosts portlets, the JBI container hosts Integration Components. The Components have to be written in Java, or can be re-used, in the same way as you write or use EJBs or portlets.

a) Concepts

The notion of "JBI component" comes with several main concepts represented by the following SPI (System Programming Interface):

- The `Component` object, with which the container interacts to retrieve information about the component (services description, etc.),
- The `LifeCycle` object, used by the container to manage the life cycle of the component,
- The `ComponentContext`, given by the container to the component, for communication with the JBI environment.

Additional concepts are:

- The `Bootstrap` object, which provides all operations required at install/uninstall time,
- The `ServiceUnitManager` object, used to manage the deploying of artefacts on a component.

A Component is packaged as an archive (a Zip or Jar file). This archive contains the classes of the component, the required libraries, and a descriptor file.

The way to plug a component into a JBI container is to use the management API provided by the container. This API allows you to provide to the container the location of your Component package. Then, the container processes the component archive and installs the Component. From the Component point of view, two different phases are defined:

- The installation phase, in which the JBI-container installs the component and plug it to the Bus,
- The execution phase, in which the JBI-container interacts with the component.

During the execution phase, the Component can consume services (exposed on the bus by other component) by sending messages to a service-provider. As a service-provider, it can accept such messages, process them, and send an answer to the consumer through the Bus.

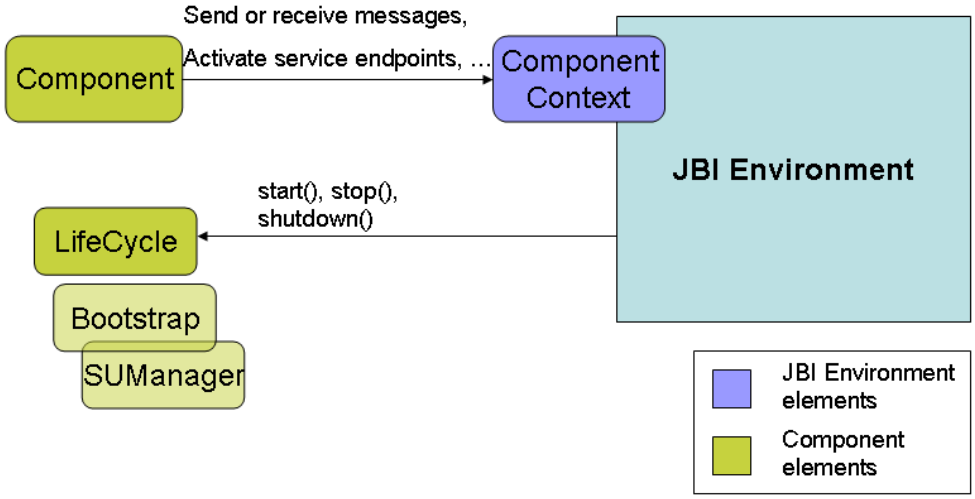


Figure 6: Interactions during the execution phase

2.4.3 Component interactions

To illustrate the interactions between a service consumer and a service provider, let's take a simple request-response exchange as example. The corresponding message exchange pattern is an "InOut" exchange pattern, as defined in the JBI specification . JBI supports four WSDL pattern exchanges: In, InOut, InOptionalOut, and RobustIn. Each pattern defines a particular exchange sequence. As an extension, it is possible to support other MEPs (Message Exchange Patterns).

a) Service consumer

Once a Component is running, it can find and consume the services that are registered in the JBI environment. It is therefore in the role of a service consumer.

Find a service endpoint

An endpoint represents an address where a service provided by a component can be found. Several components can provide the same service (with eventually different implementations), but each of those components have a unique endpoint. To find a service, the consumer asks its `ComponentContext` for the list of all endpoints matching service name, by using the `getEndpointsForService(serviceName)` method. Same methods are available to look for Interface names, or to look for a specific EndPoint via an explicit address.

Create a message exchange

To manipulate messages, the `ComponentContext` provides a `DeliveryChannel` object, which represents a bi-directional communication channel between the `Component` and the message-router of the JBI environment (called Normalized Message Router). The `DeliveryChannel` is in charge of message-exchanges instantiation, and is the path through which the messages are sent to the NMR. Then, the NMR routes the message to the component which provides the requested service.

Send the message

As soon as the consumer has instantiated a `MessageExchange`, it can set on this object the message it wants to send to the consumer. The consumer has to set a `NormalizedMessage` as the "`in`" message of the exchange. The `NormalizedMessage` is a JBI definition of a message. The consumer asks the `MessageExchange` to create a new `NormalizedMessage`. Then, the consumer sets on this `NormalizedMessage` the content of the message (an XML payload), and eventually some attachments. The consumer has to set on the `MessageExchange` the `Endpoint` of the provider (previously retrieved), and the name of the operation to be performed.
Note: the consumer can omit to specify the `Endpoint` of the provider, and just specify a `serviceName`. In this case, the NMR will search all matching `Endpoints`, and choose one of them. Finally, the consumer sends the `MessageExchange` using the `DeliveryChannel`.

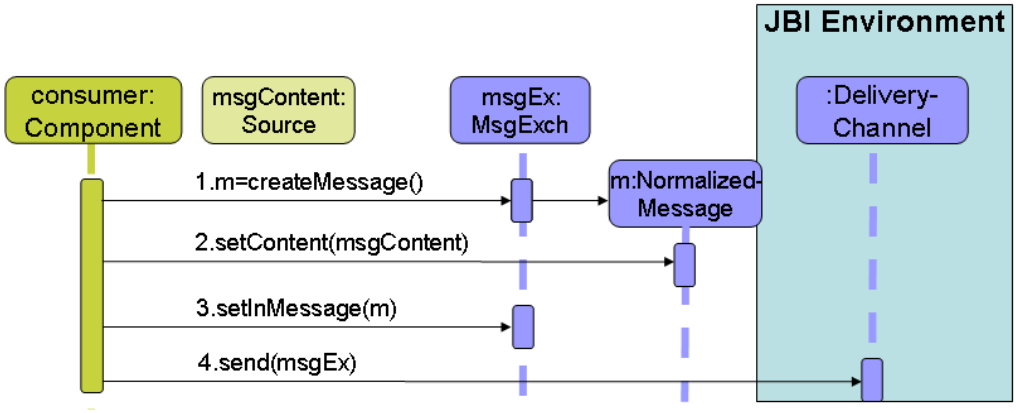


Figure 7: Send a message

b) Service provider

Once a Component is running, it can also provide services. This component acts as a service provider.

Activate an endpoint

The provider has to publish the services that it wants to offer. The publication of a service is done by the `activateEndpoint(serviceName, endpointName)` method of the `ComponentContext`. This method returns the generated `Endpoint` object that references the new service in the JBI environment. That done, other components can access the activated services, by finding the corresponding `Endpoint` with the `findEndpointForService()` method of their `ComponentContext`.

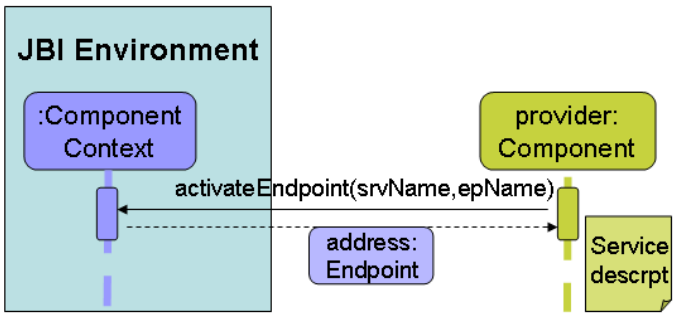


Figure 8: Activate a service - endpoint

Receive a message

Now that the provider has published some services, it can receive messages from other components (consumers). When a consumer sends a message to a provider, the message (`MessageExchange`) is pushed in the message queue of the provider's `DeliveryChannel`. The provider retrieves the received messages from the message queue with by doing `accept()` on its `DeliveryChannel`. Once the provider obtains a `MessageExchange`, it can process it. The provider gets the "in" message (the `NormalizedMessage` set by the consumer), the name of the operation to perform, the payload of the message, etc.

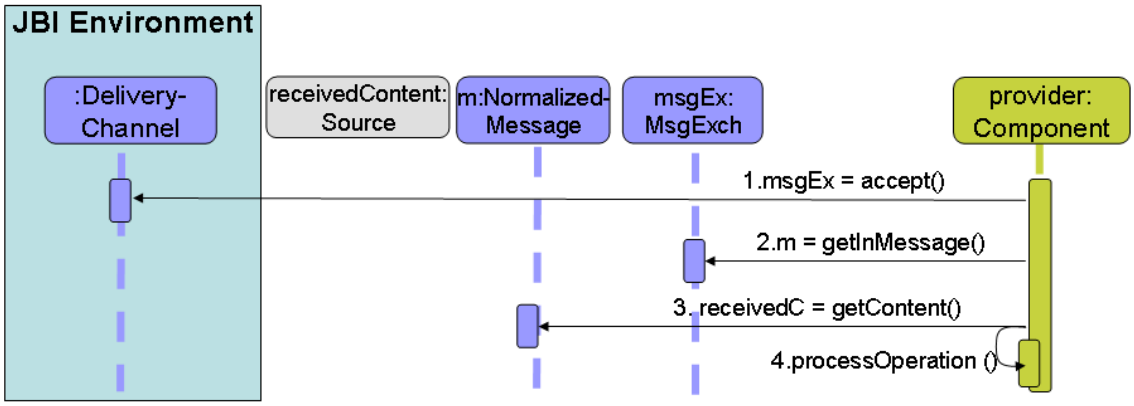


Figure 9: Receive a message

Send the response

If the operation requires an answer, the provider can set an "`out`" message on the `MessageExchange`, and send it again to the NMR, via its `DeliveryChannel`. The NMR routes the `MessageExchange` to the consumer that previously initiated this `MessageExchange`.

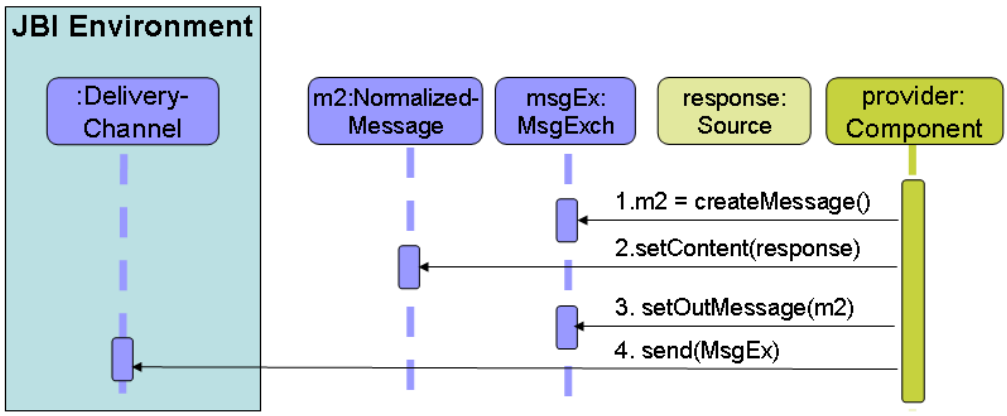


Figure 10: Send the response

Close the exchange

After the provider sent its response, the exchange is nearly complete. The NMR routes the answer to the consumer, which receives it with an `accept()` on its `DeliveryChannel`. The consumer processes the "`out`" content of the `MessageExchange`, and has to close the exchange. To do this, the consumer sets the status of the `MessageExchange` to DONE, and sends it again to the NMR via the `send()` method of its `DeliveryChannel`. The exchange is terminated, and the provider is notified of this status by receiving the `MessageExchange`. The following schema describes the whole exchange process:

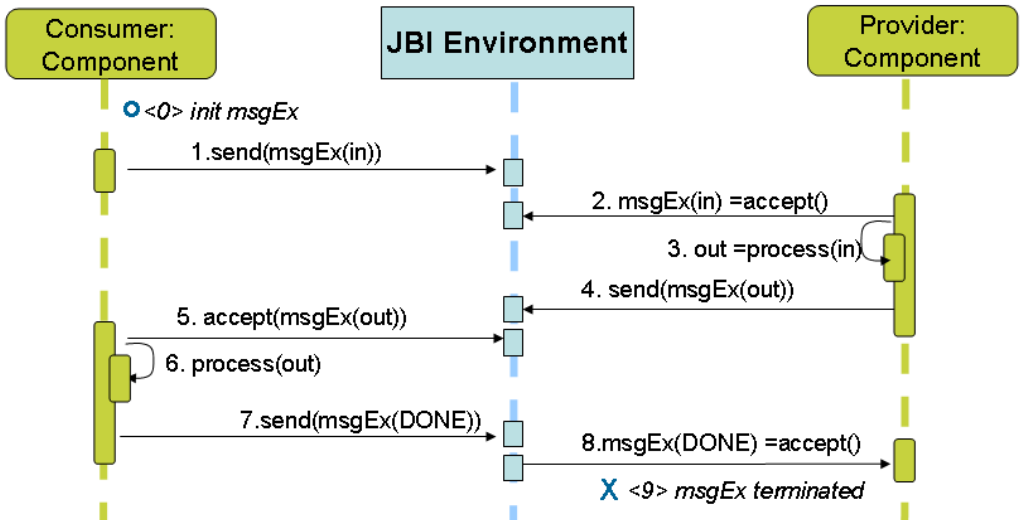


Figure 11: The whole in-out exchange

2.4.4 HelloWorldService component

As seen before, a JBI component is a set of objects that have to implement some JBI interfaces. Additionally, a descriptor file has to be provided with this component. In this section, we show most of the code to be implemented to create a simple HelloWorld ServiceEngine. As a few line of codes are necessary to implement the `Component` and the `ComponentLifeCycle` interfaces, a single object can implement these two interfaces. To process requests, there is no "listener" mechanism proposed by the JBI specification. The way to receive a message is to block on an `accept()` method. A good pattern is to create a separate object which is executed in another thread. This "listener" makes a loop on the `accept()` method, avoiding the `Component` to be blocked. The complete sources of this example can be downloaded on the Wiki page of the **Petals** project [35].

a) Component – ComponentLifeCycle

The object that implements the `Component` and the `ComponentLifeCycle` interfaces just registers the "HelloWorldService" in the JBI environment, then creates and starts the `HelloWorldListener` thread, which is in charge of processing the incoming messages.

```
import javax.jbi.component.*;
public class Helloworld implements Component, ComponentLifeCycle {
       private ComponentContext context;
       private HelloworldListener listener;
       …
       public void init(ComponentContext context) throws JBIException {
              // keep a reference to the given ComponentContext
              this.context = context;
       }
       public void start() throws JBIException {
       // create our Listener and start it.
              listener = new HelloworldListener(context.getDeliveryChannel());
               (new Thread(listener)).start();

              // register our service in the JBI environment
context.activateEndpoint(QName("http://helloWorldService.com") ,
 "HWendpoint");
       }
       public void stop() throws JBIException {
              // just break the loop of our listener; the thread will stop
              listener.running= false;
       }
       public void shutDown() throws JBIException {
```

```
                // nothing
        }
        public ComponentLifeCycle getLifeCycle() {
                // this object acts as the LifeCycle, so it returns itself
                return this;
    }
…
}
```

b) HelloWorld listener

This object processes the `MessageExchanges` that pass through the NMR. The received messages are retrieved from the `DeliveryChannel` object. This listener checks that the specified operation name is "`hello`" and that the type of the exchange is an InOut exchange. The listener retrieves the "in" content, "processes" the message (even if in this example there is no specific processing, and sets an "out" response to the `MessageExchange`. Then, it sends the `MessageExchange` back to the NMR. The response is then conveyed to the consumer through the NMR.

```
import javax.jbi.messaging.*;
public class HelloworldListener implements Runnable {
        private DeliveryChannel channel;
        public boolean running;
        public HelloworldListener(DeliveryChannel channel){
                this.channel = channel;
        }
        /**
        * the main part of the listener
        */
        public void run()
        {
                running= true;
                while(running) {
                        // block on the accept() method
                        MessageExchange messageExchange = channel.accept();
                        // a MessageExchange is received, so process it
                        process(messageExchange);
                }
        }
/**
* process the received messages
*/
        private void process(MessageExchange msg)
        {
                if( new QName("hello").equals(msg.getOperation())  && msg instanceof InOut)
                {
                        // we received a InOut exchange, with the "hello" operation
                        InOut inOut = (InOut)msg;
                        // Read the IN message
                        NormalizedMessage in = inOut.getInMessage();
                        Source content = in.getContent();
                        // … process the in-content (omitted)
                        // Write the response
                        NormalizedMessage out = inOut.createMessage();
                        // … Set out content (omitted)
                        InOut.setOut(out);
                        // Send the response
                        channel.send();
                }
        }
}
```

c) Bootstrap

No special operation is required during the installation of this HelloWorld component, so the class that implements the

`Bootstrap` interface has nothing to do.

d) Package the component

Let assume that the 3 component classes (component, listener and bootstrap) are archived in a "`helloComponent.jar`" file. A descriptor has to be provided to help the JBI container during the installation phase. This descriptor file (`jbi.xml`) looks like this:

```
<jbi version="1.0" xmlns='http://java.sun.com/xml/ns/jbi'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
       <component type="service-engine">
         <identification>
              <name>HelloworldComponent</name>
              <description>A Helloworld Component</description>
         </identification>
         <component-class-name>hello.Helloworld</component-class-name>
         <component-class-path>
              <path-element>helloComponent.jar</path-element>
         </component-class-path>
         <bootstrap-class-name>hello.Bootstrap</bootstrap-class-name>
         <bootstrap-class-path>
              <path-element>helloComponent.jar</path-element>
         </bootstrap-class-path>
       </component>
</jbi>
```

The packaging of the component is a simple archive (Zip file or Jar file) with the following structure:

- `helloComponent.jar`
- `META-INF/`
  - `jbi.xml`

e) Use the Helloworld service

A component that wants to access the service provided by this HelloWorld component can use the following code:

```
// Find the endpoint
ServiceEndpoint ep =
context.getEndpoint( new QName("http://helloWorldService.com") ,
                     "HWendpoint");

// Create the exchange
MessageExchangeFactory factory = deliveryChannel().createExchangeFactory();
MessageExchange msgEx = factory.createInOutExchange();
msgEx.setOperation( new QName("hello"));
msgEx.setEndpoint(ep);

// Create the message
NormalizedMessage nm = msgEx.createMessage();
// set the 'in' content - omitted
nm.setContent(source);
msgEx.setInMessage(nm);
deliveryChannel.send(msgEx);
```

When the HelloWorld service answers this request, the `MessageExchange` is sent back to the consumer. So, the listener of the consumer waits on the `accept()`.

```
// wait for response
MessageExchange msgEx = deliveryChannel.accept();
// process the response from helloWorldService
if( msgEx.getEndpoint=ep && msg instanceof InOut)
       {
              InOut inOut = (InOut)msg;
```

```
            // Read the OUT message
            Source content = inOut.getOutMessage().getContent();
            // … process the out-content (omitted)
            // close the exchange
            msgEx.setStatus(ExchangeStatus.DONE);
            deliveryChannel.send(msgEx);
        }
```

Note: Each `MessageExchange` created has a unique ID. To be sure to process the response of a particular exchange, the consumer can keep the reference of the sent exchange, and compare it with the ID of the received message.

2.4.5 Install and start a JBI component

A JBI container offers administrative tools to manage components, through JMX MBean objects [33].

a) Install a component

The MBean that manages the installation of components is the `InstallationServiceMBean`. This service creates, for each component to install, another MBean object, an `InstallerMBean`. The method to use is `loadNewInstaller(<archiveURL>)`. This method explodes and analyses the archive whom pathname is specified as a parameter, and returns the name of the `InstallerMBean` created. The `InstallerMBean` performs the install/uninstall mechanism, with `install()` and `uninstall()` methods. A call to `install()` instantiates the `Bootstrap` object of the component, as described in the `jbi.xml` descriptor file. Then, the `init()` and `onInstall()` methods are called on the `Bootstrap`. Finally, the `Component` object is created, and a corresponding `ComponentLifeCycleMBean` object is also created. The `install()` method returns the name of this Mbean. At this point, the component is installed, in a shutdown state.

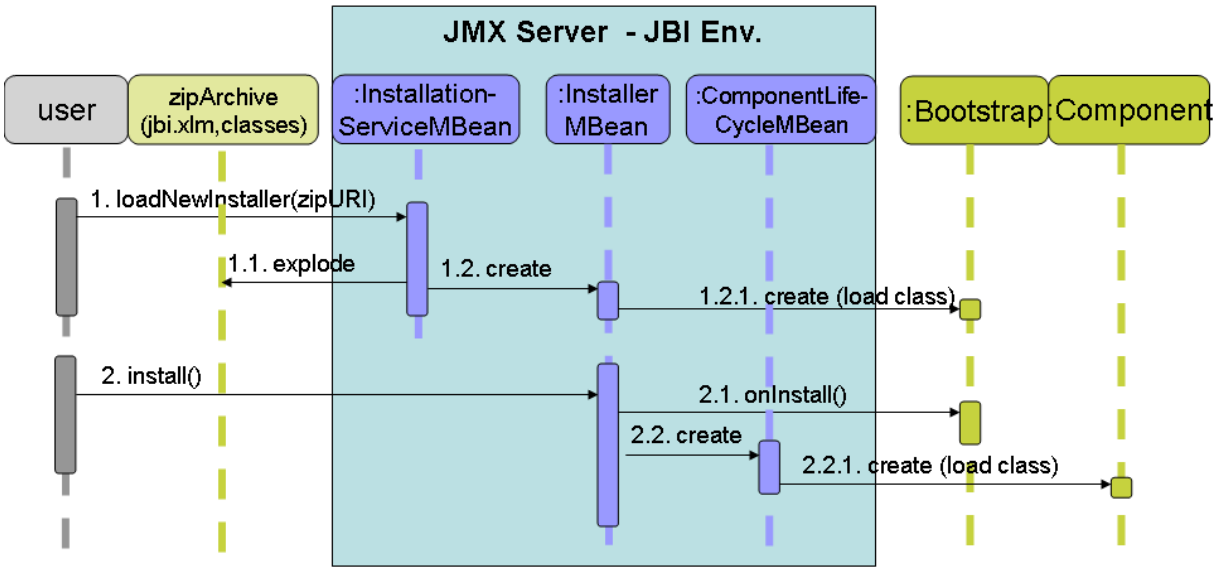


Figure 12: Install a component with JMX management

b) Start a component

The `ComponentLifeCycleMBean` manages the life cycle of the component. The main methods are `start()`, `stop()` and `shutdown()`. A call to the `start()` method of this MBean causes a `start`() on the corresponding component. If the component is started for the first time, its init() method is called before the real start.

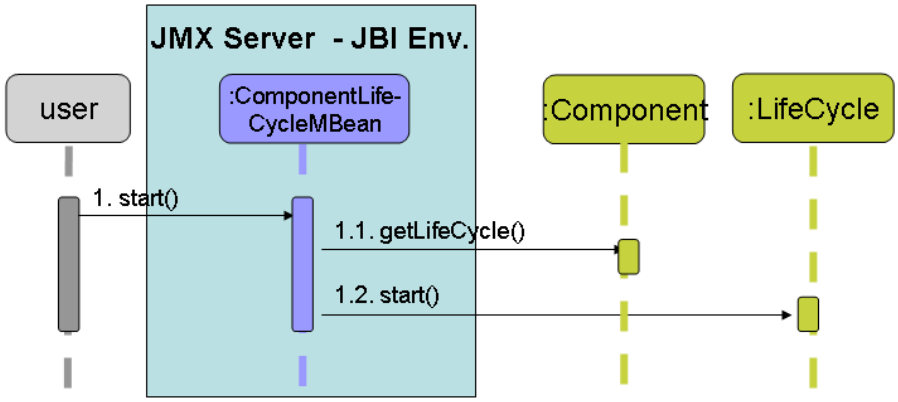


Figure 13: Start a component with JMX management

2.4.6 Conclusion

JBI standardizes how components can be managed and plugged together to create and administer "Integration platforms" built with best-of-breed integration components. The main advantages of JBI is that it standardizes :

- the Normalized Message Router for loosely coupling between components
- API for the administration of components and their configuration that enables 24x7 integration infrastructure by putting fostering on hot deployment/undeployment and on hot configuration

Clearly, JBI provides a good way to create and managed robust integration platforms. When SCA focuses on the development time, JBI focuses on the runtime.

2.5 Support SCA in JBI

2.5.1 SCA Vs JBI

JBI defines a strong integration architecture. JBI is the adequate specification to create standard based ESB solutions with a focus on agility and service oriented infrastructure. But JBI is not backed by two of the major middleware providers, IBM and BEA, certainly because it defines a concurrent technology of their proprietary middleware WebSphere ESB and Aqualogic ESB respectively. SCA defines a clean composite application model. SCA is a good way to create either composite applications or even composite services in a service oriented architecture.
As such, SCA and JBI are not opposed or overlapping technologies, but rather complementary technologies.
Next parts show how integrate the best of the two worlds in the context of the ScorWare project.

2.5.2 Design in SCA and run in JBI

Some experts think that the best interaction between SCA and JBI is to use:

1) SCA at design time to represent an SOA application or mediations
2) JBI as the runtime

This way, the tooling focuses on SCA and transformations are defined from SCA metamodel to JBI implementation. Work on this approach is being done in the context of the ScorWare project by INRIA with the STP-IM (Intermediate Model) sub project. One issue with this approach is that the two standards, don't work exactly on the same concepts.
We think that it is also interesting to use SCA to compose services and to define new tools that specifically address JBI concepts.

2.5.3 SCA as service engine

SCA is a good candidate to implement a JBI service engine specialized in service composition. In this case, SCA design tools are used to create composites and JBI handles the deployment of such composites in the SOA supported JBI infrastructure.
Indeed, by transforming every reference of a composite as a JBI reference, we can propose an additional level of flexibility because the referenced service can be modified without any change to the composite. We propose exactly the same type of interaction between JBI and a BPEL engine. For example, let's consider a BPEL engine used standalone. In this case, each service called by the BPEL process is typically accessed as a Web Service. Eventually, the BPEL engine can give the opportunity to also access JMS services. But when we integrate a BPEL engine such in a JBI environment such as PEtALS, the service referenced in the BPEL process can be provided by any JBI endpoint, meaning that the service can result in an arbitrarily complex JBI interaction. With this example, it becomes clear that we provide two complementary levels of abstraction: BPEL for process definition, and JBI for SOA infrastructure.
We apply the same principle to the relation between SCA and JBI. SCA, with it full featured tooling, eases the definition of composite services. Whereas
The integration of SCA with JBI allows to link several SCA environments via the JBI bus and to benefits from the distributed features of a JBI implementation like PEtALS to create an agile and managed SOA environment. Indeed, the references needed by a composite are resolved by the JBI environment independently of the fact that the referenced service is provided by another SCA composite, by a Web Service, by a JMS message, by a mail, or even by a file interaction.

This flexibility brought by the use of JBI and the loosely coupling provided by the NMR is the first interest of the integration of an SCA implementation inside a JBI one. Additionally, a JBI implementation like PEtALS comes with other built in advantages like a number of existing components that should be re-implemented in an SCA-centric environment, like a distributed environment, or a monitoring environment that enable the monitoring of the interactions between the SCA composites and the referenced services.

2.6 The Fractal Component Model

Fractal is an open component model that can be implemented in many different programming languages and that has been used for building various kinds of systems: OS kernels, application servers, communication libraries, transaction services, multimedia applications, etc.

The development of Fractal has been motivated by the fact that existing component-based frameworks and architecture description languages provide only limited support for extension and adaptation, as witnessed by recent works on component aspectualization, e.g. [9], [23], [1]. The paper [23] argues at length, for instance, about the lack of tailorability of EJB containers, meaning that there is no mechanism to configure an EJB container or its infrastructural services, nor is it possible to add new services to it. This limited support for extension and adaptation implies several important drawbacks: it prevents the easy and possibly dynamic introduction of different control facilities for components such as non-functional aspects ; it prevents application designers and programmers from making important trade-offs such as degree of configurability vs performance and space consumption ; and it can make difficult the use of these frameworks and languages in different environments, including embedded systems. Even with reflective component models such as OpenCOM [13], i.e. models of components endowed with an explicit meta-object protocol to control the execution of components and introduce support for different non-functional aspects, we find it necessary to be able to finely tailor the reflective capabilities endowed in components in order to support the different trade-offs which are critical in low-level software infrastructure design, e.g. in operating system or middleware construction.

The Fractal component model alleviates the above problems by introducing a notion of component endowed with an open set of control capabilities. In other terms, components in Fractal are reflective, in the sense that their execution and their internal structure can be made explicit and controlled through well-defined interfaces. These reflective capabilities, however, are not fixed in the model but can be extended and adapted to fit the programmer's constraints and objectives.

The first version of the Fractal specification was released in July 2002. This first version evolved to give birth to the second version in September 2003. This second version has never been modified. Rather, the numerous works conducted using the Fractal model shown that this specification was mature and well-suited to the design of various systems.

The Fractal component model (see [7] for a detailed specification), is a general component model which is intended to

implement, deploy and manage (i.e. monitor, control and dynamically configure) complex software systems, including in particular operating systems and middleware. This motivates the main features of the model:

- *Composite components* (components that contain sub-components), in order to have a uniform view of applications at various levels of abstraction.
- *Shared components* (sub-components of multiple enclosing composite components), in order to model resources and resource sharing while maintaining component encapsulation.
- *Introspection capabilities*, in order to monitor and control the execution of a running system.
- *Re-configuration capabilities*, in order to deploy and dynamically configure a system.

To allow programmers to tune the control of reflective features of components to the requirements of their applications, Fractal is defined as an extensible system. Control features of components are not predetermined in the model, rather the model allows for a continuum of reflective features or *levels of control*, ranging from no control (black-boxes) to full fledged introspection and intercession capabilities (including e.g. access and manipulation of component contents, control over components life-cycle and behavior, etc).

2.6.1 Components and bindings

A Fractal component is a run-time entity that is encapsulated, has a distinct identity, and that supports one or more interfaces. An *interface* is an access point to a component (similar to a “port" in other component models), that implements an *interface type* (i.e. a type specifying the operations supported by the access point). Interfaces can be of two kinds: server interfaces, which correspond to access points accepting incoming operation invocations, and client interfaces, which correspond to access points supporting outgoing operation invocations.

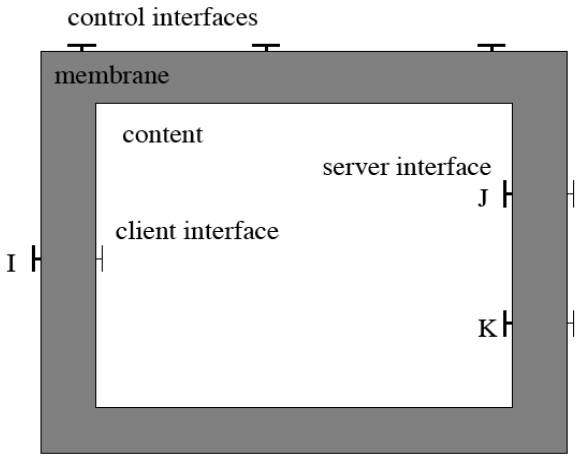


Figure 14: A Fractal Component

A Fractal component (see Figure 14) can be understood generally as being composed of a *membrane*, which supports interfaces to introspect and reconfigure its internal features, and a *content*, which consists in a finite set of other components (called *sub-components*). The membrane of a component can have external and internal interfaces. External interfaces are accessible from outside the component, while internal interfaces are only accessible from the component’s sub-components. The membrane of a component is typically composed of several controllers. Typically, a membrane can provide an explicit and causally connected representation of the component’s sub-components and superpose a control behavior to the behavior of the component’s sub-components, including suspending, checkpointing and resuming activities of these sub-components. Controllers can also play the role of interceptors. Interceptors are used to export the external interface of a subcomponent as an external interface of the parent component. They can intercept the oncoming and outgoing operation invocations of an exported interface and they can add additional behavior to the handling of such invocations (e.g. pre and post-handlers). Each component membrane can thus be seen as implementing a particular semantics of composition for the component’s sub-components. Controller can be understood as meta-objects or meta-groups as they appear in reflective languages and systems.

The Fractal model provides two mechanisms to define the architecture of an application: component nesting we have just described and bindings. Communication between Fractal components is only possible if their interfaces are bound. Fractal supports 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 (which can be modeled as a component), which means that operation invocations emitted by the client interface should be accepted by the specified server interface. A primitive binding is called that way for it can be readily implemented by pointers or direct language references (e.g. Java references). A primitive binding between a client interface c and a server interface s of two components C and S must verify one of the following constraints (see Figure 15):

- c and s are external interfaces, and C and S have a direct common enclosing component. Such bindings are called *normal bindings*.
- c is an internal interface, s is an external interface, and S is a sub component of C. Such bindings are called *export bindings*.
- c is an external interface, s is an internal interface, and C is a sub component of S. Such bindings are called *import bindings*.

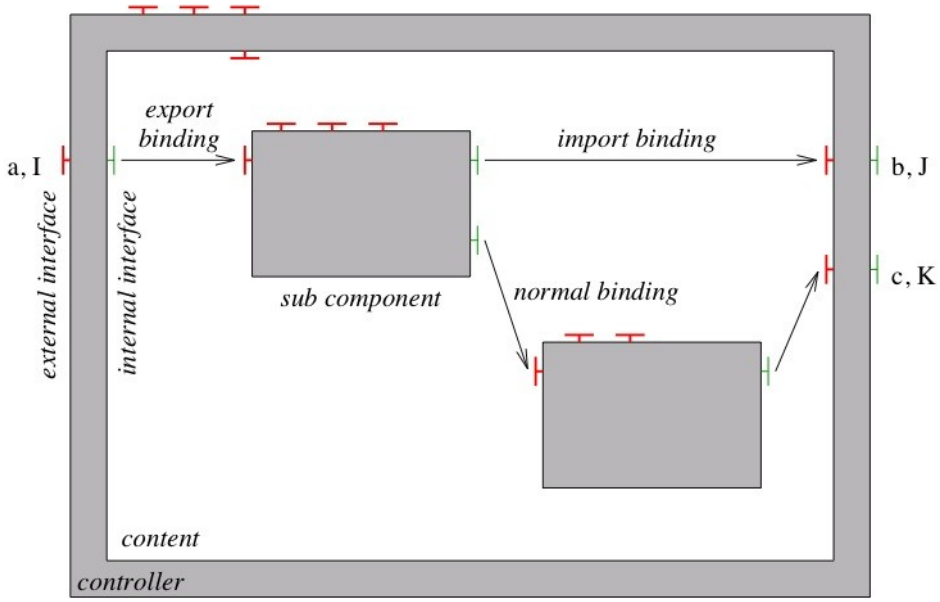


Figure 15: Fractal Component, Internal Architecture

A *composite binding* is 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).

A binding is a normal Fractal component whose role is to mediate communication between other components. The binding concept corresponds to the connector concept that is defined in other component models. Note that, except for primitive bindings, there is no predefined set of bindings in Fractal. In fact bindings can be built explicitly by composition, just as other components. Importantly, bindings can embody remote communication paths between interfaces, and span different address spaces and different machines in a network. This allows the construction of distributed configuration of Fractal components.

An original feature of the Fractal component model is that a given component can be included in several other components. Such a component is said to be *shared* between these components. Consider, for example, a menu and a toolbar components (see Figure 16 and 17), with an "undo" toolbar button corresponding to an "undo" menu item. It is natural to represent the menu items and toolbar buttons as sub components, encapsulated in the menu and toolbar components, respectively. But, without sharing, this solution does not work for the "undo" button and menu item, which must have the same state (enabled or disabled): these components, or an associated state component, must be put outside the menu and toolbar components. With component sharing, the state component can be shared between the menu and toolbar components, in order to preserve component encapsulation. Shared components are also useful to faithfully model access to low-level system resources (which are typically shared between applications), and to help separate "aspects" in component based applications (for instance it is possible to have components representing address spaces, i.e. a physical architecture, with sub components shared with other components representing a logical architecture).

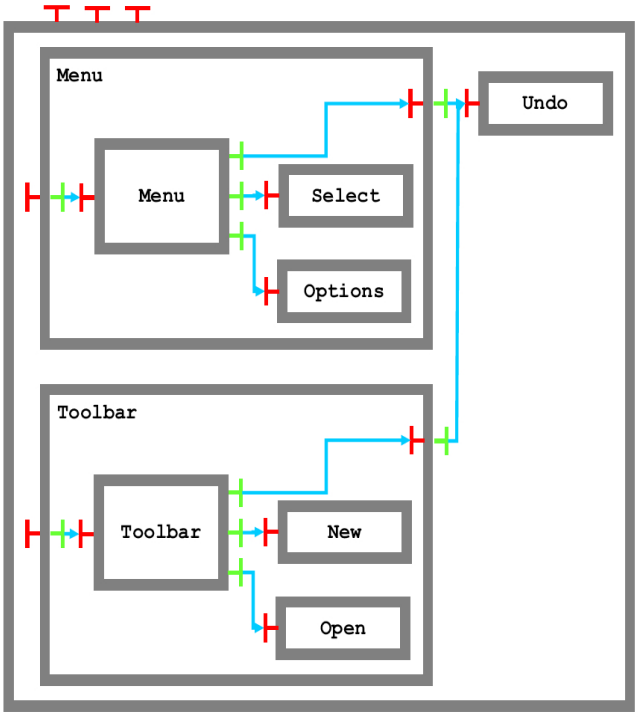


Figure 16: Architecture without shared components

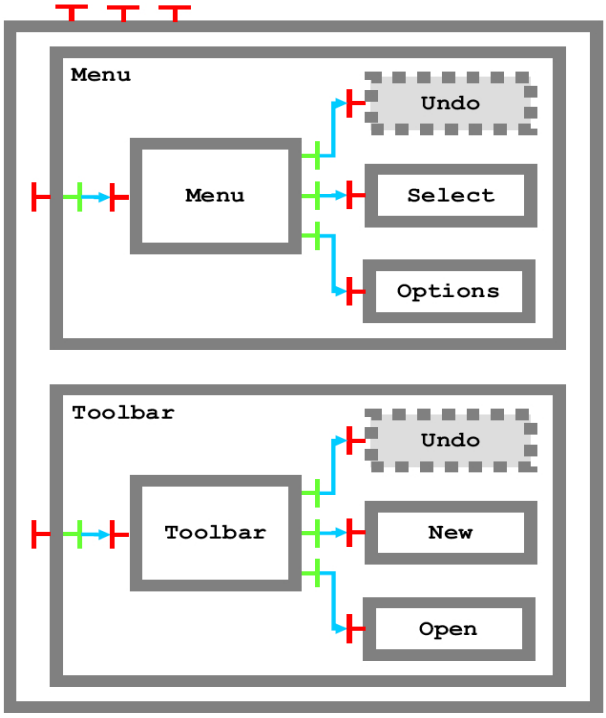


Figure 17: Architecture with shared components

2.6.2 Levels of control

The Fractal model does not enforce a fixed and pre-determined set of controllers in component membranes (hence the denomination "*open* component model"). It allows instead arbitrary forms of membranes, with different control and interception semantics. The Fractal specification, however, identifies specific forms of membranes and controllers, corresponding to different levels of control (or reflection capabilities) on components.

At the lowest level of control, a Fractal component is a black box, that does not provide any introspection or intercession capability. Such components, called *base components*, are comparable to plain objects in an object-oriented programming language such as Java (although, even at the lowest level of control, the model allows components to have a varying number of interfaces during their lifetime). Their explicit inclusion in the model facilitates the integration of legacy software.

At the next level of control, a Fractal component provides a Component interface, similar to the *IUnknown* in the COM model, that allows one to discover all its external (client and server) interfaces. Each interface has a name that distinguishes it from other interfaces of the component. At this level of control, components still do not provide any control function, but the Component interface provides elementary means for introspecting the external structure of a component. Also at this level of control, component interfaces may additionally support, via multiple interface inheritance, operations that allow retrieving the Component interface of the supporting component. Such operations are gathered in the Interface interface type.

At upper levels of control, a Fractal component can expose elements of its internal structure, and provide increased introspection and intercession capabilities. The Fractal specification provides several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features:

- **Attribute controller:** An attribute is a configurable property of a component. A component can provide an AttributeController interface to expose getter and setter operations for its attributes.
- **Binding controller:** A component can provide the BindingController interface to allow binding and unbinding its client interfaces to server interfaces by means of primitive bindings.
- **Content controller:** A component can provide the ContentController interface to list, add and remove

subcomponents in its contents.

- **Life-cycle controller:** A component can provide the LifeCycleController interface to allow explicit control over its main behavioral phases, in support for dynamic reconfiguration. Basic lifecycle methods supported by a LifeCycleController interface include methods to start and stop the execution of the component.

2.6.3 Type system

The Fractal model is endowed with an optional type system (some components such as base components need not adhere to the type system). Interface types describe the operations supported by an interface, the role of the interface (client or server), as well as its *contingency* and its *cardinality*. The contingency of an interface indicates if the functionality corresponding to this interface is guaranteed to be available or not, while the component is running:

- The operations of a *mandatory* interface are guaranteed to be available when the component is running. For a client interface, this means that the interface must be bound. As a consequence, a component with mandatory client interfaces cannot be started until all these interfaces are bound.
- The operations of an *optional* interface are not guaranteed to be available. For a server interface, this can happen e.g. when the complementary internal interface of the supporting component is not bound to a sub-component interface. For a client interface, this means that the component can execute without this interface being bound.

The cardinality of an interface type *T* specifies how many interfaces of type *T* a given component may have. A *singleton* cardinality means that a given component must have exactly one interface of type *T*. A *collection* cardinality means that a given component may have an arbitrary number of interfaces of type *T*. Such interfaces are typically created lazily, e.g. upon request of a bind operation through a BindingController interface.

Component types are just sets of component interface types. The type system is equipped with a subtyping relation which embodies constraints to ensure substitutability of components.

2.6.4 Instantiation

The Fractal model also defines *factory* components, i.e. components that can create new components. Again, the Fractal model does not constrain the form and nature of factory components, but the Fractal specification provides useful forms of such factories. In particular, it distinguishes between *generic component factories*, which can create several kinds of components, and *standard factories*, which can create only one kind of components, all with the same component type. A generic factory provides the GenericFactory interface, which allows a new component to be created, given its type, and an appropriate description of its membrane (controllers) and content. A *template* is a special standard factory that creates components that have the same internal structure as the template. Thus, a template component can have several templates as sub-components (sub-templates). A component created by such a template will have as many sub-components as sub-templates in the template, which will be bound together in the same way as the sub-templates are. Templates are useful to manifest at run-time a particular configuration.

2.7 Mapping SCA to Fractal

The SCOrWare platform provides a runtime implementation for SCA-based applications. This implementation is composed of two main parts: the FraSCAti assembly factory and Tinfi. The assembly factory is in charge to interpret SCA composite descriptors and to instantiate SCA components. These components are running on top of the Tinfi runtime environment built on top of the Fractal component model. Then the combination of the FraSCAti assembly factory and Tinfi implements a mapping of SCA assembly concepts to Fractal runtime concepts. This mapping is listed in Table 3. For instance, SCA components are mapped to Fractal components. SCA services and references are mapped to Fractal server and client interfaces, respectively. SCA properties are mapped to Fractal component attributes.

SCA Concepts	Fractal Concepts
Component	Component
Composite	Composite component
Implementation	Component content
Component with a composite implementation	Composite component
Component with a Java implementation	Primitive component
Component and composite name	Component name
Property	Attribute
(Both composite and component) Service	Server interface
(Both composite and component) Reference	Client interface
Interface	Interface type
Interface multiplicity	Cardinality and contingency
Wire	Normal binding
Service promotion	Export binding
Reference promotion	Import binding
Binding of a service	Server interface exported by the Binding Factory
Binding of a reference	Client interface bound by the Binding Factory

Table 3: Mapping SCA to Fractal concepts

2.8 Summary

This chapter has discussed the state of the art related to the SCOrWare project, and presented the SCOrWare vision. Our vision is that both architecture-based and service-oriented approaches must be unified in a common framework. SCA is the current industrial standard that promotes such a common component model for building SOA applications. But in parallel, JBI is another industrial component standard to build SOA integration platforms. Finally, Fractal is a general purpose component model that has been successfully used to build various operating systems, middleware, containers, and application servers. Then the SCOrWare runtime platform is based on these three models. SCOrWare implements a platform for deploying and running SCA applications. This SCOrWare platform, named FraSCAti, is deployed on top of the OW2 PEtALS JBI container. Finally, both PEtALS and FraSCAti are implemented on top of the Fractal component model.

3 Common Meta Model between Platform and Tooling

This chapter contains specifications related to the Task 1.2 of the SCOrWare project. This is dedicated to the design of the SCOrWare Meta Model based on the assembly model specification [14]. Section 3.2 describes the development process from the OSOA schema to the common meta model of the SCOrWare platform and tooling. Then, Section 3.3 specify the SCOrWare Meta Model which correct the SCA assembly model, and defines validation rules for this Meta Model.

3.1 Objectives

The SCA assembly model [14] is the center part of the platform and the tools provided by the SCOrWare project. The SCA assembly structure is defined by a set of XML Schema Definitions (XSD) given by the SCA assembly model specification. These schema allow us to load, handle, validate and save SCA assembly definitions with the help of existing XML tools. On one hand, the SCOrWare platform can use the XML schema to parse and verify SCA assembly correctness before running composite, while on the other hand the SCOrWare tools use the schema to generate assembly descriptions. Our objective in this part is to design a common meta model between the SCOrWare platform and the tooling. We have particularly studied the SCA specification and the XML schema, and then report correction to obtain schema coherent with the specification. We also extracted a list of rules described in the assembly model specification. These rules define constraints which have to be verified when designing an SCA assembly. With the corrected schema and the list of constraints on the SCA model, we have started an SCA meta model based on the Ecore model and the Eclipse Modeling Framework [31].

3.2 Architecture

The XML schema given in the SCA specification can be considered as the default implementation of the SCA assembly model specification. In the section 3.3, we identify errors in these XML schema and provide corrections in order to have the schema coherent with the specification. In the context of the SCOrWare project, we choose to describe the SCA assembly model using the Eclipse Modeling Framework (EMF) [31]. In this section we explain how the SCA EMF model is obtained.

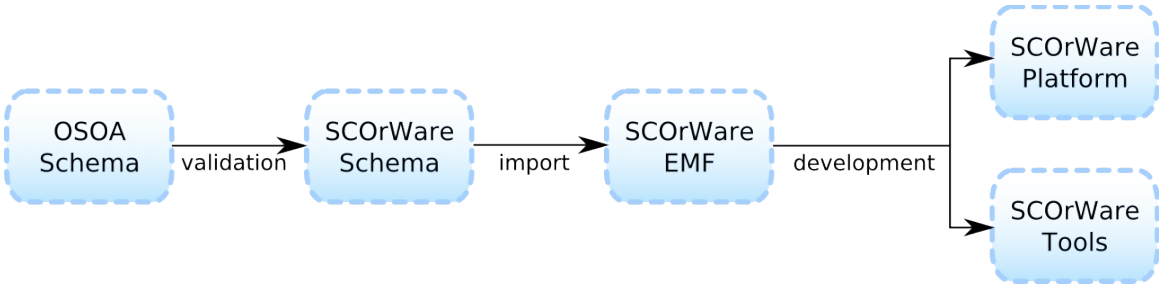


Figure 18: From XSD resources to SCA meta model

While some SCA compliant platforms provide direct implementation of SCA concepts (Apache Tuscany SCA for instance), the SCOrWare project, through the Eclipse Modeling Framework, considers a model oriented approach to design the platform and associated tools. Section gives details about choosing EMF, but we can remind that using such modeling tools help us in focusing on the SCA model rather than details of implementation. The framework offers code generation tools which facilitate the developer tasks and improve code maintenance. Moreover it is able to generate basic editor and validator for a model.

EMF is able to import XML schema definition and generate the corresponding model into the framework. At this point, the specification of the SCOrWare XML schema, which result from corrections given after, becomes relevant to EMF : in order to be able to generate a model, it needs validated XML Schema. Using the corrected schema, EMF generates a meta model of the SCA specification. This meta model reifies the SCA concepts defined in the schema as concepts of the Ecore model (The EMF meta model). Finally, the SCA Ecore model needs to be edited to add some informations which cannot be specified with the XML schema. For instance, it is better to define reference from SCA service to SCA reference with the `Ereference` concept of the Ecore model, rather than the `string` attribute defined in the schema. To

sum up, we have created our own SCA model for SCOrWare. This model is obtained in two steps (Fig. 18): (i) first step, the schema correction from the provided OSOA schema which results in the SCOrWare schema, (ii) second step the schema importation to the Eclipse Modeling Framework which results in the SCOrWare model.

The Eclipse Modeling Framework, with our SCA meta model, can also generate implementation code to read, write, edit assembly descriptions. It means that the meta model can be used both for generating implementation which allows to read SCA assembly description and instantiate components, and generating editor which permits to design SCA assemblies.

3.3 Specification

The SCA assembly model specification describes a set of XML Schema Definition (XSD) files that defines the language of SCA assembly descriptions. XML Schema Definition [28] provides a means for defining the structure, content and semantics of XML documents. XML Schema definitions allow to describe XML vocabularies for a XML document class. In the context of SCA, XSD provides data types, elements, their attribute and the structure of an architecture description language (ADL) to describe SCA assemblies. In the next sections, we study the SCA model defined by the set of XSD files provided by the SCA specifications.

3.3.1 Schema validation

Giving a first look at the XSD files which define the SCA model, we observe that the files cannot be validated according to the XSD specification (tested via the XML Schema Validator available at W3C website [26]). The following errors are detected for schema defining the model :

- Some resources schema are not found
- Attempt to extend with an attribute or an element already declared
- Non-deterministic content model (violate the “Unique Particle Attribution” rule)

Such non-validated model can introduce errors when parsing and handling assembly definitions. We need to ensure that XSD files reflect correctly the SCA Assembly Model Specification, especially for discussing solutions for processing assembly descriptions, and to verify their structural properties. As a contribution to the SCA assembly model specification, we propose some corrections to enable validation of the model. We have listed these corrections according to each XSD file.

a) Correction in `sca-core.xsd`

The file `sca-core.xsd` defines the model for the main concepts of the Service Component Architecture described in the assembly model specification. In this XML schema definition file, the complex types Reference, Service, ComponentReference and ComponentService cause the schema to be non-deterministic. This error is caused by the element `choice` :

```
<choice minOccurs="0" maxOccurs="unbounded">
  <element ref="sca:binding" />
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
</choice>
.
.
<any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
```

We observe that the XML elements `any` used twice are in conflict. Actually the element `choice` is useless. This part can simply be replaced by defining an element of type `binding` which has 0 or more occurrences. The correction applied between lines 54-61, 78-83, 202-208, and 228-233 is written below.

```
<element ref="sca:binding" minOccurs="0" maxOccurs="unbounded" />
.
.
<any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
```

The name of imported composite is required for the `include` elements. This is not currently described in the

schema, and thus, the `use` attribute of the `include` element has to be set to required (schema line 264).

```
<complexType name="Include">
  <attribute name="name" type="QName" use="required" />
  <anyAttribute namespace="##other" processContents="lax" />
</complexType>
```

Note : SCA components and composites can be configured via external properties. According to the specification, a property requires a value for attribute `type` or `element`. The definition of an SCA property is not correct in the schema. But, because choice between two attributes can not be expressed with XSD, this constraint cannot be verified with the schema.

b) Correction in *sca-implementation-java.xsd*

The schema definition given by the file `sca-implementation-java.xsd` extends the schema defining main SCA concepts in order to describe components implemented with Java. This definition tries to override attributes (line 20 & 21) which is not allowed in XML schema definition.

```
<attribute name="requires" type="sca:listOfQNames" use="optional"/>
<attribute name="policySets" type="sca:listOfQNames" use="optional"/>
```

The attributes `requires` and `policySets` are already defined in the `sca-core.xsd` schema. The definition of these attributes in `sca-implementation-java.xsd` need to be deleted.

c) Correction in *sca-implementation-composite.xsd*

Like Java implementation, the file `sca-implementation-composite.xsd` adds definition of composites implementation to the SCA model. The attributes `requires` and `policySets` are also defined, and need to be deleted (lines 19 & 20).

d) Correction in *sca-binding-sca.xsd*

The definition in `sca-binding-sca.xsd` adds description of SCA bindings to the SCA model by extending the `binding` type.

```
<complexType name="SCABinding">
    <complexContent>
        <extension base="sca:Binding">
          <sequence>
            <element name="operation" type="sca:Operation" minOccurs="0"
                     maxOccurs="unbounded" />
          </sequence>
          <attribute name="uri" type="anyURI" use="optional"/>
          <attribute name="name" type="QName" use="optional"/>
          <attribute name="requires" type="sca:listOfQNames"
                use="optional"/>
          <attribute name="policySets" type="sca:listOfQNames"
                use="optional"/>
           <anyAttribute namespace="##any" processContents="lax"/>
        </extension>
    </complexContent>
</complexType>
```

In this schema, the SCA binding overrides elements and attributes of a binding defined by the SCA core model. Redefining a type is not allowed in the XML schema definition norm. Thus, the element `operation` and the attributes `uri, name, requires` and `policySets` are useless in this definition. As a result an SCA binding is just a renaming of a binding defined in the SCA core model.

```
            <complexType name="SCABinding">
              <complexContent>
                <extension base="sca:Binding">
                  <anyAttribute namespace="##any" processContents="lax"/>
                </extension>
              </complexContent>
            </complexType>
```

e) *Correction in `sca-binding-webservice.xsd`*

The `sca-binding-webservice.xsd` schema extends the model with description of web service bindings. Web service binding defines how an SCA assembly can be exposed as a web service and/or invoke web services. As an SCA binding, a web service binding extends the binding definition from the SCA core model. The web service binding definition we use is :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
xmlns:wsdli="http://www.w3.org/2004/08/wsdl-instance"
xmlns:wsa="http://www.w3.org/2004/12/addressing"
elementFormDefault="qualified">
  <import namespace="http://www.w3.org/2004/08/wsdl-instance"
  schemaLocation="wsdli.xsd" />
  <import namespace="http://www.w3.org/2004/12/addressing"
  schemaLocation="ws-addr.xsd" />
  <include schemaLocation="sca-core.xsd" />
  <element name="binding.ws" type="sca:WebServiceBinding"
  substitutionGroup="sca:binding" />
  <complexType name="WebServiceBinding">
    <complexContent>
      <extension base="sca:Binding">
        <sequence>
          <element ref="wsa:EndpointReference" minOccurs="0"
          maxOccurs="unbounded" />
          <any namespace="##other" processContents="lax"
          minOccurs="0" maxOccurs="unbounded" />
        </sequence>
        <attribute name="wsdlElement" type="anyURI" use="optional" />
        <attribute ref="wsdli:wsdlLocation" use="optional" />
        <anyAttribute namespace="##any" processContents="lax" />
      </extension>
    </complexContent>
  </complexType>
</schema>
```

However, the definition provided by the specification document has some errors. First, as schema for SCA policy, imported schemes are wrong. We correct these references (lines 9 to 12) with the following input.

```
<import namespace="http://www.w3.org/2004/08/wsdl-instance"
         schemaLocation ="http://www.w3.org/2004/08/wsdl-instance" />
<import namespace="http://www.w3.org/2005/08/addressing"
        schemaLocation="http://www.w3.org/2006/03/addressing/ws-addr.xsd" />
```

Second, the possible extension of the web service binding type causes the model to violate the unique particle attribution rule of the XML schema definition norm. We need to delete possibility to extend this binding type (schema line 22) to ensure strict validation of the schema. This is demonstrated below :

```
        <extension base="sca:Binding">
          <sequence>
            <element ref="wsa:EndpointReference" minOccurs="0"
             maxOccurs ="unbounded"/>
            <any namespace="##other" processContents="lax" minOccurs="0"
             maxOccurs ="unbounded"/>
          </sequence>
          <attribute name="wsdlElement" type="anyURI" use="optional"/>
          <attribute ref="wsdli:wsdlLocation" use="optional"/>
          <anyAttribute namespace="##any" processContents="lax"/>
        </extension>
```

f) Correction in `sca-policy.xsd`

The SCA policy document defines constructs to express non-functional requirements associated to SCA components such as quality of service (QoS), capabilities or constraints. This document is based on the Web Services Policy Framework (WS-Policy) and Web Services Policy Attachment (WS-PolicyAttachment) [27] specification submitted to the W3C. The reference to the imported WS-Policy schema (line 11) is wrong in the SCA policy model.

```
    <import namespace="http://schemas.xmlsoap.org/ws/2004/09/policy"
        schemaLocation="http://schemas.xmlsoap.org/ws/2004/09/ws-policy.xsd"/>
```

The schema location must be corrected with the following input.

```
 <import namespace="http://schemas.xmlsoap.org/ws/2004/09/policy"
     schemaLocation="http://schemas.xmlsoap.org/ws/2004/09/policy/ws-policy.xsd"/>
```

g) Correction in `sca-definitions.xsd`

The SCA specification defines variety of artifacts which are not specific to a particular component or composite. These artifacts are defined in a file named `definitions.xml` that conforms to the schema `sca-definitions.xsd`. Actually these artifacts include intents, policy sets, bindings, binding type and implementation type. The specification declares also a `targetNamespace` attribute, missing in the schema. This attribute needs to be added after definition of shared artifact. The description below gives the correction to apply after line 20 of the schema.

```
<element name="definitions">
  <complexType>
    <choice minOccurs="0" maxOccurs="unbounded">
      <element ref="sca:intent"/>
      <element ref="sca:policySet"/>
      <element ref="sca:binding"/>
      <element ref="sca:bindingType"/>
      <element ref="sca:implementationType"/>
      <any namespace="##other" processContents="lax" minOccurs="0"
      maxOccurs="unbounded"/>
    </choice>
    <attribute name="targetNamespace" type="anyURI" use="required"/>
  </complexType>
</element>
```

Applying those corrections allow us to validate SCA model schema against the XML schema definition norm. The new schemas provided by SCOrWare contribute to the SCA specification by giving a model definition which can be used to verify correctness of SCA assembly descriptions.

3.3.2 Model structure

After correcting the SCA specification schemas in order to validate them, we now analyze the model structure. When trying to validate SCA assembly samples (from SCA specifications and Tuscany distribution [2]) with the new model definition, we discover an error in relation with the model structure. The model definition is valid according to XSD norm, while it didn't reflect the specification written in the SCA assembly model document. As a result we cannot validate the SCA assembly using the XML schema definition. SCA assembly seems to have errors because the model used to load them in memory is wrong. Like for the schema validation, we propose a modification to conform the schema to the specification.

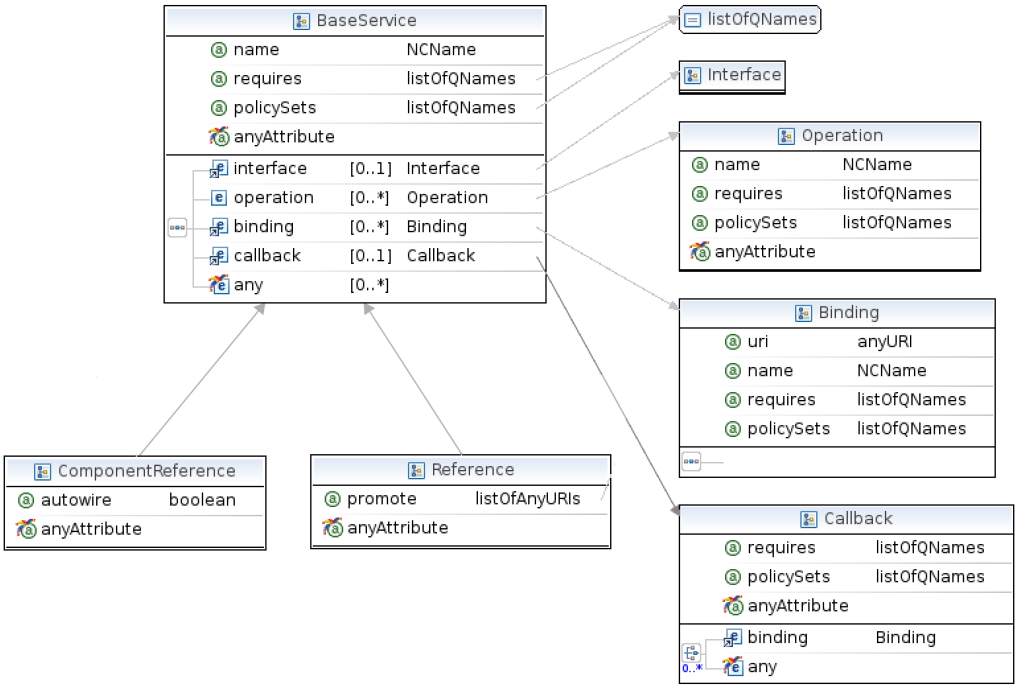


Figure 19: Definition of a basic reference

The definition provided by the SCA assembly model specification makes distinctions between a reference in a composite and a reference in a component. The former promotes a component contained in a composite while the later describes the operations required by a component. The schema defines a component reference (noted `ComponentReference)` as a restriction of a composite reference (noted `Reference)`. The difference between a `Reference` and a `ComponentReference` is symbolized by a `promote` attribute. This attribute refers to a component which is promoted by the composite. It is defined as a required attribute for a `Reference`.

Extending by restriction a `Reference` implies that the attribute `promote` is needed in the extension. Thus, the attribute `promote` is also required in a `ComponentReference`. However, as described in the specification (SCA Assembly model, page 5) a reference or a service, in a component, has not a `promote` attribute. This points out a structural problem in the definition between a composite reference and a component reference. Composite reference and component reference have few differences. Instead of specifying a component reference as a restriction of a composite reference, we can define a composite reference and a component reference as an extension of a basic reference type (Fig. 19). A reference extends this type by adding an attribute `promote` while a component reference adds an attribute `autowire`. Similarly, service expressed in a composite or service expressed in a component have the same problem. We also define `Service` and `ComponentService` as extensions of a basic service type.

In this part we reveal that SCA model as defined by the SCA schema comprises some errors which need to be corrected. We have argued each correction for each problem detected. The SCA schema proposed by SCOrWare are validated according to the XSD norm. The new schema allows, in scope of SCOrWare, to perform SCA assembly validation. Actually, the XML schema definitions define a grammar for the SCA architecture description language. This means that the core of the SCOrWare platform will be able to detect errors when loading SCA assemblies. This capability is currently not supported by other SCA platforms like Tuscany SCA. A drawback of using XSD models is every extensions of the SCA specification (binding protocol, implementation...) must be also described with XSD. The schema conform to the SCA specifications are given at the end of this document.

3.3.3 Validation Rules

The SCA assembly model specification defines the concepts used in Service Component Architecture, the Architecture Description Language (ADL) written in XML form to describe SCA assemblies according to its model, and gives rules related to concepts of the model. These rules written along the specification defines constraints on assembly descriptions and its implementation imposed by the SCA model. Thus a valid SCA application must verify these rules. The objective of this part is to list the rules defined by the SCA assembly model specification. However an SCA assembly definition succes in the schema validation test, it doesn't mean this definition is a valid SCA assembly. As a result we have to take into account the rules written in the specification in order to ensure correctness of SCA assembly definitions. We distinguish here two different parts : (i) the rules which apply on the assembly definition and the rules which verify coherence between description and implementation.

a) Verify assembly description

In the next table, we list rules which apply on assembly descriptions. We identify each rule according to the SCA concept it refers to and where, in the SCA assembly model document, the rule is defined. These rules are complementary to the syntax of a composite document.

SCA Concepts	Lines	Rules
Component	142	Component names are unique in a composite.
Reference	203, 211, 1581	Multiple target services are valid when the reference has a multiplicity of `0..n` or `1..n`.
Composite, Composite Reference	220,1351	When `WiredByImpl` attribute is `true`, then the reference should not be wired statically.
Wire	1630	Wire link sources and targets contained in the same composite when composites are used as component implementations.
Wire	1621	URI scheme for source attribute is `<component-name>/<reference-name>` where the source is a component reference. The reference name is optional if the source component only has one reference.
Wire	1625	URI scheme for target attribute is `<component-name>/<service-name>` where the target is a service of a component. The service name is optional if the target component only has one service with a compatible interface.
Wire	1634	Source and target interfaces must be compatible.
Wire	1648	If connected interfaces are written with different interface languages (ex: WSDL and Java), operations defined by the two interfaces must be equivalent (interface mapping).
Composite Implementation	1852	The internal construction of the composite is invisible to the using component. The using component can only connect wires to the services and references of the used composite and set values for any properties of the composite.
Composite Implementation	1860	The composite must have at least one service or at least one reference.
Composite Implementation	1864	Each service offered by the composite must be wired to a service of a component or to a composite reference.
Composite Reference, Composite Service	1325, 1498	Names are unique in a composite.
Composite Reference, Composite Service	1061, 1059	Composite references/services involve the promotion of one or more references/services of one or more components.
Composite Reference, Composite Service	1328, 1501	Promote attribute value is a list of values of the form `<component-name>/<reference-name>` (`<component-name>/<service-name>`) separated by spaces. The reference/service name is optional if the component has only one reference/service.
Composite Reference Composite Service	1358, 1512	Composite reference/service interface must be compatible with promoted component interface (Same or superset).
Composite Reference	1364	Multiplicity attribute must be compatible with multiplicity of component reference.
Composite Reference	1392	Multiplicity attribute of component reference must be `0..n` or `1..n` when promoted by multiple composite references.
Composite Reference	1395	A composite reference can promote multiple components when : • Components have same interfaces or Composite reference interface is compatible with component interfaces • Multiplicities of component are compatible • Intents of components are compatible
Binding	2307	Uri attribute is required for references defined in composites contributed to SCA domains.
Binding	2319	When a service or a reference as multiple bindings, only one can have the default name value (the service or reference name); all others must have a value specified

		that is unique within the service or reference.
Constraining Type	2182	An implementation constrained by a constraining type must define services, references, and properties specified by the constraining type. The implementation cannot contain additional non optional references or properties.
Constraining Type	2188	When a component is constrained, the containing composite can only see a projection of the component type.
Constraining Type	2196	When a constraining type includes required intents, those intents are applied to any components that use this constraining type.
Composite Property	1100	A composite property requires to specify one of `type` or `element` attribute.
Composite Inclusion	1991	The composite resulting from inclusion must be a valid composite.

Table 4: Assembly Model Rules

b) Verify coherence between assembly and implementation

The Service Component Architecture is divided in many documents. While the assembly model specification defines mains SCA concepts, the Java implementation document describes how to create SCA applications with the Java language. In fact, assembly description and component implementation in SCA are correlated. This implies that assembly and implementation must be coherent. For instance, reference names in the assembly description and in the implementation have to match. The next table sums up what we have to check between assembly description and implementation.

SCA Concepts	Lines	Rules
Service, Reference, Property	164,192, 279	The name of a service/reference/property has to match the name of a service/reference/property defined by the implementation.
Service, Reference	177,226	If an interface is specified it must provide a compatible subset of the interface provided by the implementation.
Property	274	The property type specified must be compatible with the type of the property declared by the implementation.

Table 5: Implementation and assembly model coherence rules

3.4 Summary

In this section, we have focused our attention on studying the Service Component Architecture specifications in order to establish the SCOrWare meta model. The analysis of the specifications reveals that the SCA schema are erroneous. Since they are invalid, the SCA schema becomes useless, that's why we describe how the schema must be corrected. Thus, our first contribution to the SCA community relies in the release of the correction and the resulting schema. Currently, this work has been submitted to the Eclipse STP community and the Open SOA collaboration in order to improve the specification. All the schemas used in SCOrWare are listed in annex. The XML schema provided by SCOrWare defines the structure of SCA assembly documents while they respect the XSD norm. Moreover, the SCOrWare schema allow to create the SCA meta model using EMF. Our meta model of the SCA specification is the common part of the SCOrWare platform and its tools. Both of them takes benefit from the model driven approach and the Eclipse Modeling Framework which facilitates their development process. Additionally, we have extracted constraints on the assembly model from the specifications. These rules must also be verified when designing or loading SCA assemblies.

4 Runtime Environment

This chapter contains specifications related to the Task 1.3 of the SCOrWare project. This is dedicated to the design and implementation of the SCOrWare runtime environment to host and execute SCA Java-based components. This runtime is split into two parts: Tinfi and its integration with JBI. Section 4.1 describes Tinfi as our Fractal-based implementation of the both SCA "Java Common Annotations and APIs" and "Java Component Implementation" specifications. Section 4.2 describes how Tinfi is integrated into the PEtALS JBI container platform.

4.1 Tinfi

4.1.1 Objectives

Tinfi is the SCA runtime platform of the SCOrWare project. The name Tinfi stands for "Tinfi Is Not a Fractal Implementation".
At the date of the writing of this document, three other open source runtime platforms exist for the SCA component model: Tuscany SCA [2] from the Apache Foundation, Fabric3 [5] from the Codehaus foundation and Newton [4] from the Cauldron community. Yet, these three platforms are black-box implementations: they are seldom extensible and they are build from scratch with the Java object-oriented language, mostly ignoring the corpus of existing work around component-based software engineering and middleware engineering.
Our objective with the Tinfi runtime platform for SCA is to provide an environment where the component containers can be easily extensible with services for e.g. remote communications management, semantical trading, deployment, reconfiguration or transaction management.
This objective is achieved by founding Tinfi as a personality of the existing Fractal [7] component model. We take advantage of the genuine feature of Fractal which allows customizing the control part of a software component. By doing this, we are able to change the semantics of the non functional properties of a component. Thus, we obtain software components which are both Fractal-compatible and SCA-compatible. This allows reusing the corpus of Fractal technical and middleware services which are already existing, or which are to be defined in the context of SCOrWare such as the Binding Factory (see Section 5).

4.1.2 Specification

a) Requirement

Our main requirement is to provide a runtime platform for executing applications written as assemblies of SCA Java components. An SCA Java application is defined by the Open SOA Collaboration [34] as a set of software components which implement the OSOA API and which are assembled with the so-called Assembly Language. The Assembly Language is addressed in Chapter 3 and Section 8.1 of this document. This section focuses on the OSOA API, and more specifically on the Java version of this API which is defined in the document titled "SCA Service Component Architecture Java Common Annotations and APIs" [18] version 1.0 released in March 2007.
This document defines a set of rules for an SCA/Java application. This is mandatory for a Java program to enforce these rules to be a legal SCA application. The goal of the Tinfi platform is to interpret (note: to be understood as interpret in the general sense, not by opposition with to compile) these rules and to perform the corresponding behaviours which are defined in the SCA/Java specification. This specification takes the form of an API along with the semantics of this API defined in English. This API heavily uses the new features of the Java 5 language, and among others, annotations and generics. More precisely, this API is composed of:

- 5 interfaces plus an additional interface containing only constants,
- 4 exception classes,
- 25 annotations which can be classified into 3 groups:
  - 16 general-purpose annotations for defining components, required and provided services,
  - 2 annotations for remote communications,
  - 7 annotations for capturing non functional properties related to security, more precisely, confidentiality, integrity, authentication.

b) Validation

Two types of action will be undertaken to validate the fact that Tinfi meets this requirement. The first action will be to ensure that applications developed for other existing SCA platforms are correctly executing when run under Tinfi. For that, we will take the sample applications which are available for the Tuscany SCA [2] platform. The second action will be to develop a set of tests. These tests will be developed independently of the platform and will materialize the behaviours which are defined in the OSOA specification as plain English.
Note that this second action will be an added-value of the Tinfi platform with respect to the other existing platforms and with respect to the OSOA Collaboration as well. Indeed, as surprising as it may seem, the OSOA specification is not defined more formally than an English text. There is, for example, no such thing as an SCA TCK (Technology Compatibility Kit) as this is the case for Java EE. There is, up to this day, no solution for certifying an SCA platform. The unitary tests that we propose in this action will be a way of bootstrapping this. These tests will not be tied to Tinfi and will be reusable for any other SCA platform.

4.1.3 Architecture

Tinfi is architectured as a so-called personality of the existing Fractal [7] component model and its reference implementation, the Julia framework. A personality is an extension of the model/framework meant to provide different flavours of software components. The purpose of Tinfi is thus to extend Fractal/Julia to obtain SCA software components. Tinfi also uses some aspect-oriented programming (AOP) [10] notions to associate intent handlers to SCA components.
The remainder of this section introduces the basic notions of the Fractal/Julia framework which are needed to present the architecture of the Tinfi runtime platform, proceeds by presenting the architecture of the Tinfi platform, its control interfaces and implementations and its interceptors.

a) Basic notions of the Fractal/Julia component framework

Fractal is a hierarchical (with sharing), reflective and open software component model. The model is primarily meant to implement middleware but is general enough to accommodate the needs of other domains, such as the one of graphical components. The model is independent of the programming languages and several languages are supported: Java, C/C++, SmallTalk, C# and the languages of the .NET plaftorm. We focus here on the Java version of Fractal and on its reference implementation, the Julia framework.
In existing component models such as EJB, SCA, OSGi, CCM or .NET, software components are hosted in a so-called container which is responsible for providing the extra-functional services (e.g. security, transaction management, data persistence) needed by the application. Contrary to these models, Fractal does not assume that there is a fixed and immutable set of such services. Rather, the founding principle of the approach is to let these technical services be customized, added, removed, etc. in order to tailor the applications to various execution contexts. Hence, Fractal can be seen as providing a solution for hosting software components with open containers which can be programmed. For this, four main notions (control membrane, control interface, controller and interceptor) are available.

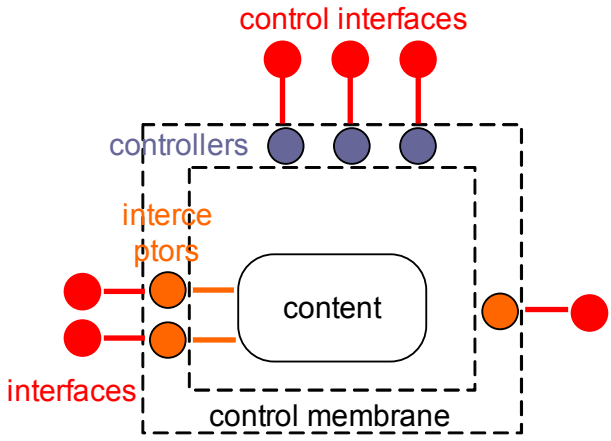


Figure 20: Structure of a Fractal component

The structure of a Fractal component is illustrated in Figure 20. A Fractal component is composed of a **content**. As the model is hierarchical, the content is either other subcomponents or, at the leaves of the hierarchy, an entity of a programming language such as a Java object, which implements the business logic assigned to the component. A component owns some externally accessible access point which are called **interfaces**. Interfaces are either incoming (server interfaces for provided services) or outgoing (client interfaces for required services). Two categories of interfaces are considered: business interfaces and control interfaces. As their name suggests it, business interfaces are concerned with the business logic. The **control interfaces** are the access points to the extra-functional services provided to the components. These extra-functional services are implemented in a **membrane** which is composed of controllers and interceptors. A **controller** is the basic unit of implementation of an extra-functional service. In most cases, a controller implements a control interface. However, this is not mandatory and a controller may implement several control interfaces or none. An **interceptor** intercepts the business communication flow to add the logic which may be needed by an extra-functional service.

b) *Tinfi runtime platform*

Tinfi is implemented as a personality of the Fractal/Julia framework. This means that some controllers, some control interfaces and some interceptors have been specified and implemented to match the OSOA API specifications [18]. As illustrated in Figure 21, the Tinfi runtime platform is thus composed of a set of controllers providing extra-functional services to the components of an SCA application. This application consists of a set of components wired by their interfaces. For completeness sake, one should mention that the interceptors mentioned in the figure are generated by the platform.

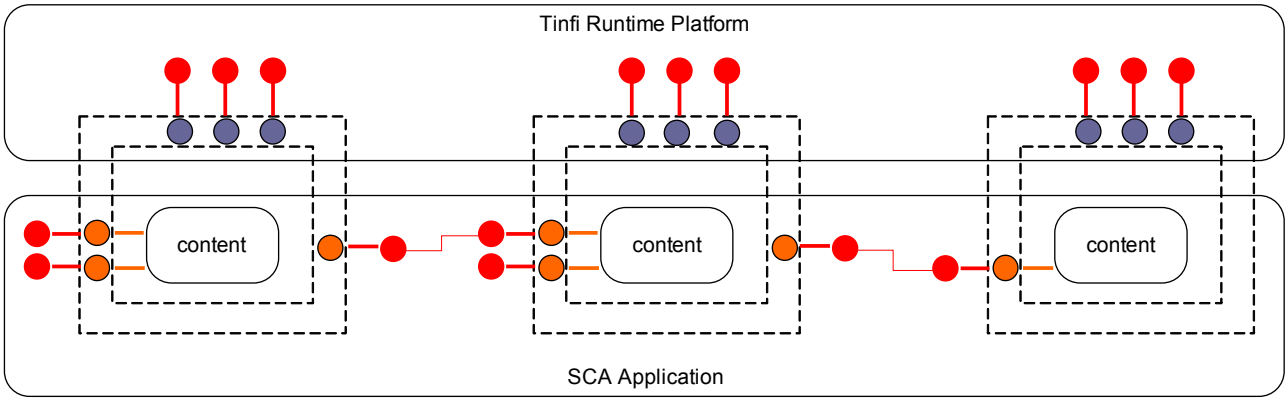


Figure 21: Tinfi runtime platform

c) *Tinfi control interfaces*

Two control membranes are provided by Tinfi: `scaPrimitive` and `scaComposite`. The former corresponds to a component which is implemented with a Java class, whereas the latter is a component whose content consists of other (sub)components. By extension, we designate under the term `scaPrimitive (resp. scaComposite)` any component equipped with a `scaPrimitive (resp. scaComposite)` membrane.

Nine control interfaces are associated to a `scaPrimitive` component. The first four interfaces are specific to `scaPrimitive` components.

- `sca-component-controller`

  This control interface supports the SCA component identity of a `scaPrimitive` component. This interface implements the `org.osoa.sca.ComponentContext` type which is defined in the OSOA API. See [18] for the list of methods defined by this type.

- `sca-content-controller`

  This control interface manages the content of a `scaPrimitive` component. The SCA component model defines the notion of scoped-components with four possible values: `stateless` (the component is stateless), `request` (a component instance per request), `conversation` (a component instance per conversation with

a client), `composite` (a component instance per containing top-level component). The `sca-content-controller` is in charge of managing the SCA component instances according to these policies. This control interface is used internally by the interceptors and is not meant to be manipulated by some peer components or controllers of the platform. The following methods are available with this interface:

- `Object getFcContent() throws InstantiationException`: Returns a content instance according to the scope policy defined for this component. Throws `InstantiationException` if the content can not be instantiated.
- `void releaseFcContent( Object content, boolean isEndMethod )`: Notifies the content controller that the specified content instance is no longer needed. If relevant with the scope policy, this gives the opportunity to the controller to call the `@Destroy` annotated method on the content instance. `Content` is the content instance which is released. `IsEndMethod` values to `true` if the method which releases the content instance is annotated with `@EndsConversation`.
- `Stack<RequestContext> getRequestContextStack()`: Returns the thread-local stack of request contexts.
- `void setRequestContextStack( Stack<RequestContext> stack )`: Sets the thread-local stack of request contexts.
- `void eagerInit() throws InstantiationException`: Eager initializes the content instance associated with this component. Relevant only for composite-scoped components. Throws `InstantiationException` if the content can not be instantiated.

- `sca-property-controller`

  This control interface manages the properties of an SCA component. The following methods are available with this interface:
  - `void set( String name, Object value )`: Sets the value for the specified property name. If the property has already been set, the old value is lost, and the new value is recorded.
  - `Object get( String name )`: Returns the value for the specified property name. Return `null` if the property has not been set.
  - `boolean containsKey( String name )`: Returns true if the specified property name has been set.
  - `Map<String,Object> getAll()` : Returns all property names and values.
  - `void setPromoter( String name, SCAPropertyController promoter )`: Sets the reference of the property controller which promotes the specified property to the current property controller.
  - `SCAPropertyController setPromoter( String name )`: Returns the reference of the property controller which promotes the specified property. Returns `null` if the property is managed locally by the current property controller.

- `sca-intent-controller`

  This control interface manages the intent handlers associated with an SCA component. The following methods are available with this interface:
  - `void addFcIntentHandler( IntentHandler handler )`: Adds the specified intent handler on all service and reference methods of the current component.
  - `IntentHandler[] listFcIntentHandler()`: Returns the list of intent handlers associated with the current component.
  - `void removeFcIntentHandler( IntentHandler handler )`: Removes the specified intent handler from the current component.

The five remaining control interfaces are Fractal/Julia standard interfaces [6].

- `binding-controller`: Manages the binding between the services required by the component and the services provided by peer components. The following methods are provided:

- `String[] listFc()`: Returns the list client interface names associated with this component.
- `void bindFc( String name, Object dst)`: Binds the dst server interface to the name client interface.
- `Object lookupFc( String name )`: Returns the server interface bounds to the name client interface.
- `void removeFc( String name )`: Removes the binding for the name client interface.

- `lifecycle-controller`: Manages the life cycle of the component. The following methods are provided:
  - `void startFc()`: Starts the component.
  - `void stopFc()`: Stops the component.
- `component`: Manages the Fractal identity and the structure of the component. The following methods are provided:
  - `Object getFcInterface( String name )`: Returns the interface named name.
  - `Object[] getFcInterfaces()`: Returns the array of interfaces belonging to the current component.
  - `Type getFcType()`: Returns the type of the current component.
- `super-controller`: Manages the component hierarchy by allowing to navigate in parent components. The following methods are provided:
  - `Component[] getFcSuperComponents()`: Returns the array of super components.
- `name-controller`: Manages the name of the component. The following methods are provided:
  - `String getFcName()`: Returns the name of the current component.
  - `void setFcName( String name )`: Sets the name of the current component.

Nine control interfaces are associated to a `scaComposite` component. The three four interfaces are specific to `scaComposite` components: `sca-component-controller`, `sca-property-controller` and `sca-intent-controller`. These interfaces are the same than the one described previously. The remaining six control interfaces are standard Fractal/Julia interfaces for composite components: `binding-controller`, `lifecycle-controller`, `component`, `super-controller`, `name-controller` and `content-controller` (the first five are the same than the ones described previously and `content-controller` manages the subcomponents inserted into the composite).

The content-controller control interface provides the following methods:

- `void addFcSubComponent( Component component )`: Adds the specified subcomponent.
- `Component[] getFcSubComponents()`: Returns the array of subcomponents.
- `void removeFcSubComponent( Component component )`: Removes the specified subcomponent.
- `Object getFcInternalInterface( String name )`: Returns the internal interface named name.
- `Object[] getFcInternalInterfaces()`: Returns the array of internal interfaces.

The fact that `scaPrimitive` (resp. `scaComposite`) components provide both some control interfaces which are specific to SCA and some which are standard to Fractal means that `scaPrimitive` (resp. `scaComposite`) components are both SCA and Fractal compliant. In other terms, a `scaPrimitive` (resp. `scaComposite`) can be assembled with other regular (non-SCA) Fractal components, while keeping its dual SCA personality.

d) Tinfi controller implementations

As mentioned previously, a control interface is implemented by a controller. Among the eight above mentioned control interfaces, five own an implementation which is specific to Tinfi: `sca-component-controller`, `sca-content-controller`, `sca-property-controller`, `binding-controller` and `lifecycle-controller`. For the three other control interfaces (`component`, `super-controller` and `name-controller`), the standard Fractal/Julia controllers are reused.

The nine controllers of a Tinfi `scaPrimitive` membrane are assembled together as illustrated in Figure 22. The

control semantics specified by the SCA model is not implemented by nine isolated entities but is the result of the cooperation of these entities. They define some provided and some required services which are wired together. They form the architecture of the Tinfi platform.

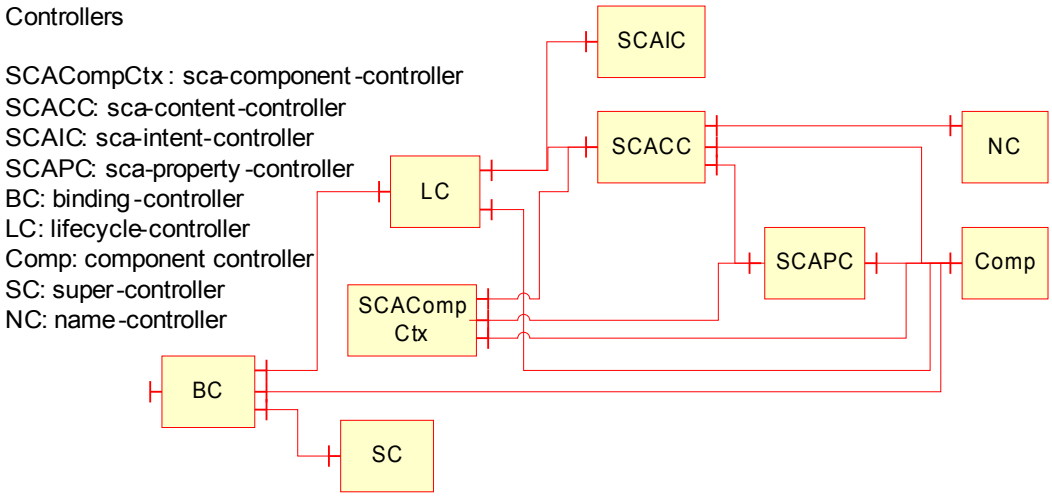


Figure 22: Architecture of a scaPrimitive membrane

By the same way, the nine controllers of a Tinfi `scaComposite` membrane are assembled together as illustrated in Figure 23.

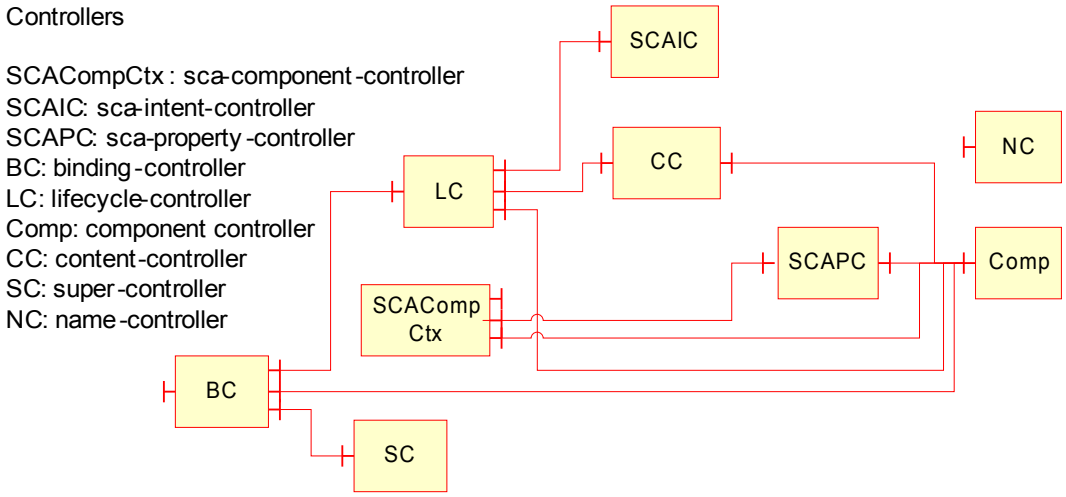


Figure 23: Architecture of a scaComposite membrane

e) *Tinfi interceptors*

As mentioned previously, a control membrane is composed of controllers and interceptors (interceptors are not mandatory). Three interceptor types are defined by the Tinfi platform.

- `lifecycle-interceptor`: This is the standard Fractal/Julia interceptor. Its role is to apply the life cycle policy onto business interfaces. More precisely, this interceptor blocks operation invocations of a stopped component and releases them once the component is started. The purpose is thus to ensure that a software

architecture can be reconfigured by safely stopping components.

- `sca-content-interceptor`: This interceptor is specific to Tinfi. This interceptor collaborates with the `sca-content-controller` to manage the SCA component instances according to the scope policy defined by the component developer. The dynamics of this interceptor is illustrated in Figure 24. An operation invocation arriving on an incoming interface is delegated to the `sca-content-interceptor`. This interceptor forwards the request to the `sca-content-controller`. According to the defined scope policy, this controller decides either to create a new instance or to reuse an existing instance. For example, with conversation-scoped components, there is one such instance per current conversation with a client, whereas, for request-scoped components, there is one instance per request. The reference of the instance is returned to the `sca-content-interceptor`.

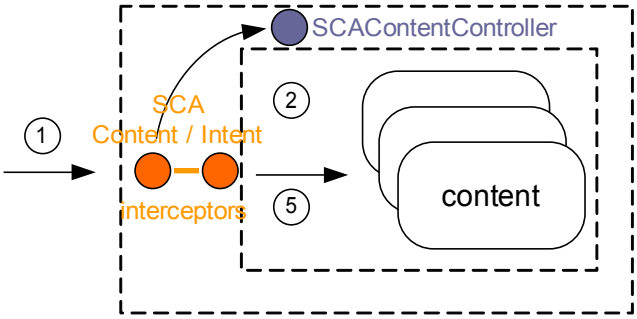


Figure 24: SCA content interceptor

- `sca-intent-interceptor`: This interceptor is specific to Tinfi. This interceptor implements an aspect-oriented programming (AOP) [10]style of development for gluing non functional services to SCA components. The idea, illustrated in Figure 25, is to program non functional services as regular SCA components. The advantage of doing this is to keep an uniform formalism for programming both the business logic and the non functional logic of the application. This idea has been pioneered in AOKell [12] for Fractal components and transferred here to SCA.

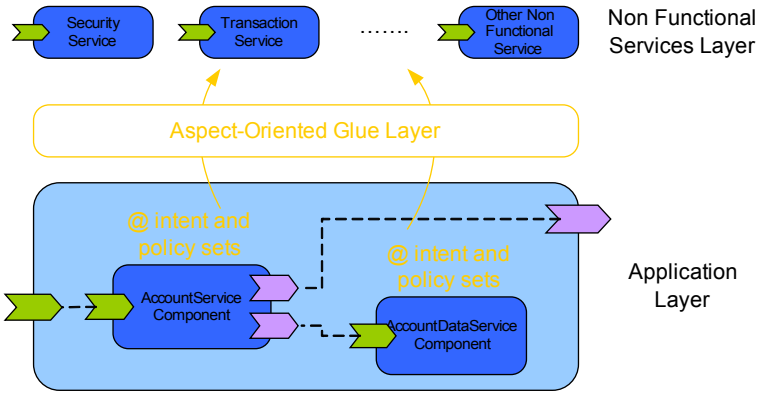


Figure 25: SCA intent management

The `sca-intent-interceptor` collaborates with the `sca-intent-controller` to manage the SCA intent handlers defined by the component developer. The dynamics of this interceptor is illustrated in Figure 26. An operation invocation arriving on an incoming interface is delegated to the `sca-intent-interceptor`. This interceptor forwards the request to the `sca-intent-controller` (step 3 in Figure 26) which forwards the request to the first element in the list of intent handlers associated with the current component (step 4). The intent handler executes its before code, and proceeds with the next handler in the list or if this is the end of the list, by delegating the invocation to the content instance (step 5). Note that if the list is empty, the `sca-`

`intent-controller` skips step 4 and proceeds directly by delegating the invocation to the content instance (step 5). Once the method has been executed, the after blocks of code defined in the list of intent handlers are executed in reverse order (step 6).

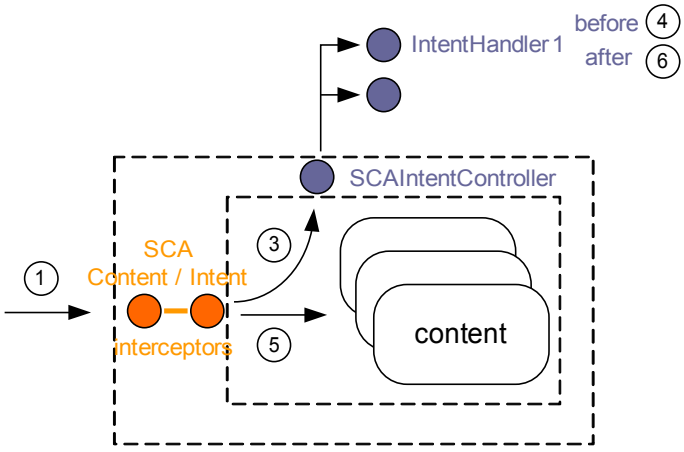


Figure 26: SCA intent interceptor

4.1.4 Implementation

The implementation of the Tinfi runtime platform reuses the tools which exist for the Fractal/Julia framework. This is thus a matter of configuring this framework with the appropriate pieces of code. As previously mentioned, two parts can be isolated in a Fractal component: the content which implements the business logic and the control which implements the extra-functional properties. Some glue code is needed to bind these two parts together. Currently, two techniques exist for this gluing:

- at runtime with the Julia framework and the ASM [30] bytecode engineering library,
- at compile-time with the Juliac code generator (Juliac is available from http://fractal.ow2.org/juliac).

Note that both techniques are not in opposition and that they can be used jointly in a same application, some parts being glued at compile-time, while others are glued at runtime. In each case, some code generators need to be developed to implement the proper gluing: bytecode generator in the case of Julia, and Java source code generators in the case of Juliac. The Tinfi platform currently provides the Java source code generators. The bytecode generators have been left for future work. Yet, no particular difficulty is expected as this is a matter of translating the generation operations from source code into bytecode.

4.1.5 Summary

This section presents the specification, the architecture and the implementation of the Tinfi runtime platform for SCA components. Tinfi provides an implementation of the OSOA API and can run applications developed as SCA/Java components. Tinfi has been developed by extending the existing Fractal/Julia framework. A set of controllers and control interfaces are proposed to implement the SCA functionalities defined in the OSOA specification. Furthermore, the SCA components which are provided by Tinfi are also compatible with standard Fractal/Julia components. This enables the connection of these components to existing applications or middleware services developed with this technology.

4.2 Integration with JBI

4.2.1 Objectives

The objective is to integrate Frascati, the SCOrWare SCA Runtime, in PEtALS, the OW2 integration platform, based on JBI and supported by EBM WebSourcing.
Several points must be addressed to successfully integrate Frascati with PEtALS and more generally JBI:

- First, Frascati must be packaged as JBI Service Engine
- SCA bindings must be able to be translated to links with the JBI binding components (JBI semantic)
- A generic JBI binding (in SCA semantic) should be added in order to simply indicate a reference to other components through the JBI bus

The fact that Frascati is included in PEtALS makes it supported in a robust runtime environment. It is also an important addition to PEtALS as SCA and JBI both get large momentum in the integration community, and as SCA adoption is being considered in the JBI 2.0 expert group. Last, it brings PEtALS to a new group of ESB that supports SCA, like WebSphere ESB for which IBM promotes the creation of mediation with SCA (see for example http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp).

4.2.2 Specification

a) PEtALS JBI Component Development Kit

The Component Development Kit (CDK) is a development framework for JBI components. It contains a set of classes to ease the development of components that kindly interface with the JBI container. It respects all the contracts between the component and the container for deployment, management and runtime environment as defined in the chapter “Component Framework” of the JBI specification.
To create a component, a developer just has to extend an abstract class according to the component type (Binding Component or Service Engine), and to extend listeners to handle the service invocations coming from the JBI Normalized Message Router (NMR).
The CDK provides a Service Unit (SU) manager which coordinates the deployment of Service Units with the deployment of CDK listeners.
For each new SU deployment, a listener is created and can access the configuration contained in the extensions part of the SU deployment descriptors.
The CDK contains utilities classes to handle easily Message Exchange content and it coherency against Message Exchange Pattern (InOnly, InOut, ...).
Last, it adds the concept of interceptors for pre and post processing of Message Exchanges by the component.
All the PEtALS Service Engines and Binding Components are managed in the same way and the component framework provides an out of the box support of the components life cycles.

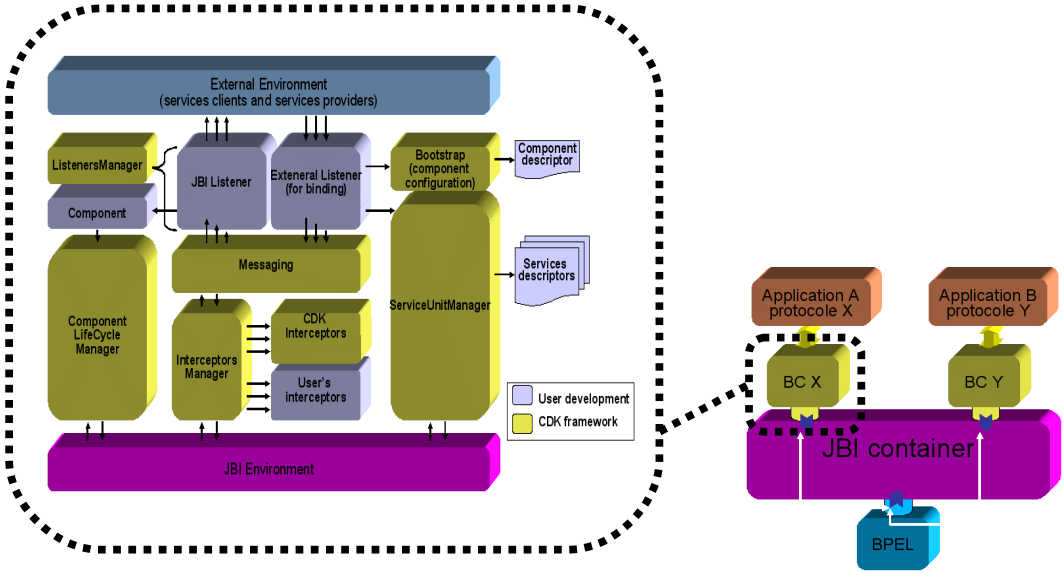


Figure 27: PEtALS JBI Component Development Kit

To use a PEtALS Component on the PEtALS ESB, the process is the following:

1. Install the component on the PEtALS server with a static configuration contained in the component descriptor.
2. Configure the component via JMX if necessary
3. Start the component.

Then, to deploy services on this component :

1. Create a Service Unit and pack it in a Service Assembly
2. Deploy the Service Assembly on the PEtALS server
3. Start the Service Assembly to expose the Service in the bus or consumes a service exposed on the bus

b) *Frascati service engine*

Packaging FraSCAti in PEtALS is all about writing the JBI component around FraSCAti.We have to deal with the usual aspects of a component as written in the previous section. Both the installation and the configuration of FraSCAti in PEtALS are something standard with the component Framework. For configuration, we have to expose FrasSCAti properties (not composite or component properties, but the runtime properties) through JMX as JMX is the preferred way for component configuration in JBI. This type of configuration is fully supported by the component framework.
The Frascati Service Engine enables integration of Frascati in PEtALS. Features of this component are:

- Deploy SCA composite embedded in a JBI Service Unit and using the standard PEtALS hot deployment and configuration mechanisms.
- Instantiate SCA components in Frascati during deployment.
- Expose SCA composite to the NMR
  - Unmarshall incoming message exchanges to Java method call.
  - Marshall returning results in message exchanges.
- Call distant references of the SCA composite as JBI endpoints.

Such integration implements delegation of PEtALS the management of binding references in SCA composites. Such a delegation conforms to the “spirit” of SCA as it decouples component implementations from bindings: component implementations are run by the SCA runtime, the Frascati Service Engine, whereas binding are bound by PEtALS binding components.

To deploy the SCA composites in the Frascati/PEtALS platform, the user must provide the following artifacts:

- Java class binaries for the component implementation

- SCA composite description (*.composite)
- WSDL of the composite services

The schema bellow shows the deployment process of the SCA composite into PetALS/Frascati.

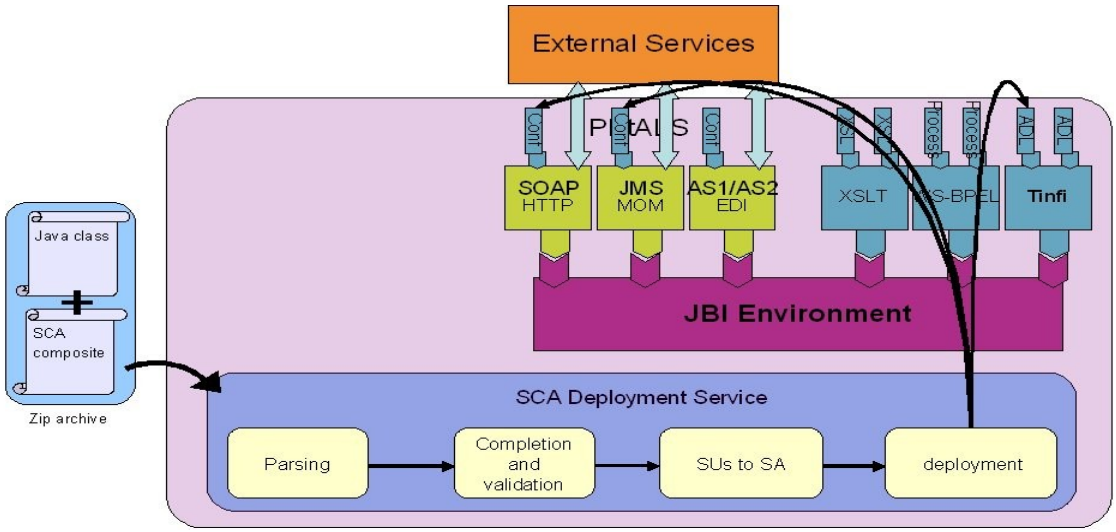


Figure 28: Composite deployement into PetALS

The JBI artifacts to configure the Frascati SE and the creation of a package to deploy in PEtALS can be generated with the PEtALS Eclipse plugin.

c) Dealings with binding

Bindings are the means by which composites can be inter-connected in SCA. SCA defines how a binding must be expressed in an SCA composite description, and it standardizes the way to express bindings for JMS, Web Services and EJB/RMI.
In JBI, Binding Components are components whose role is to create interfaces between the JBI bus and the rest of the world. In other words, JBI binding components provide the capability to :

- expose an external service (accessible through any protocol like Web Service, JMS, Mail, ...) as a endpoint in the JBI bus,
- expose JBI services (any service accessible through the JBI NMR as an external service).

With the notion of Binding Components, JBI provides the capability to interface to any service accessible by any protocol. As an example, PEtALS proposes Binding Components for JMS, Web Services, Mail, FTP, HTTP, JDBC.
With the integration of FraSCAti in PEtALS, we define JBI binding for SCA. This binding mimics WS binding and explicitly defines that the binding refers to a service accessible through a JBI bus.
With this binding, we ease the description of composites that are designed to be run in a JBI compliant environment so that these composites benefit of all the agility of JBI. By the agility of JBI, we mean that a service referenced by the SCA composite can be implemented by any of the BC or SE available in the bus.
With the binding, we provide a closer integration between the SCA design, and the SCA/JBI runtime. We can say that it enables to compose services with SCA inside the JBI infrastructure. Below the definition of the JBI binding.

```
<fra:binding.jbi
  interfaceName="" <!-- the interface name -->
  interfaceNamespace="" <!-- the interface name space -->
```

```
serviceName="" <!-- the service name -->
serviceNamespace="" <!-- the service name space -->
endpointName="" <!-- the endpoint name -->
operationName="" <!-- the operation name (optional) -->
operationNamespace="" <!-- the operation name space (optional) -->
wsdl="" <!-- the wsdl file path -->
mep="" /> <!-- the message exchange patern {InOnly, InOut, InOptionnalOut, RobustInOnly}(optional) -->
```

d) Translation of standard SCA bindings to JBI connections

During the deployment of a composite in PEtALS, the Frascati deployment service transforms every distant binding described in the composite as JBI bindings.
The main advantage of this transformation is that the references defined in the composite can then be provided by any service exposed in JBI regardless of its implementation as soon as it provides the right interface. This type of configuration enables smooth evolution of the implemented SOA configuration and typically the replacement of a referenced service by another one, eventually accessed via another protocol, simply by updating the JBI configuration.
Intra composite communications are local to Frascati as long as the composite is deployed on a unique Frascati Service Engine.

4.2.3 Summary

In the scope of SCOrWare, FraSCAti is integrated in PEtALS in a Service Engine to enable the deployment of SCA composites into the leading OW2 integration platform, and the creation of JBI components with SCA.
It is worth mentioning that by integrating FraSCAti in PEtALS, we fully address the stake of the task 1.3 as Frascati supports the SCA for Java specification and its integration in PEtALS puts it in a robust and open infrastructure.
By integrating FraSCAti in PEtALS, we also get advantage of the existing JBI binding components.
By defining a new JBI binding for SCA, we extend the SCA specifications to leverage JBI loosely coupled architecture.

5 Binding Factory

This chapter contains specifications related to the Task 1.3 of the SCOrWare project. This is dedicated to the design and implementation of the SCOrWare runtime environment to host and execute SCA Java-based components. In this section we present the objectives and the architecture for the Fractal Binding Factory project (aka Fractal-BF, or BF for short), a core module of the FraSCAti runtime platform, to establish bindings between components running in distant machines. In the reminder of this chapter, architectural and implementation details refer to the latest release of the BF at the time of this publication.

5.1 Objectives

Definition

Binding is the process of interconnecting a set of objects in a computing system. The result of this process, i.e., the association, or link, created between the bound objects, is also known as binding. The purpose of binding is to create an access path through which an object may be reached from another object. Thus a binding associates one or several sources with one or several targets [24].

The objectives of a *binding factory* are to establish and administer bindings of a certain type. A *distributed binding* is a binding established between objects running in different machines. Clients of the binding factory must specify the type of the bindings to be established and administered. The objectives of the Fractal-BF are primarily to provide such functionalities to component-based applications built on top of the Fractal component model, particularly in the context of distributed bindings.

To establish a (distributed) binding, a two-step process must be executed by the binding factory: the first step is to *export* an object as a service available for access by client according to a specific binding protocol; the second step is to bind some client object to the exported service according to a specific binding protocol. These two steps can be potentially executed independently by different parties. As an example, consider some web service published by some company (e.g., a weather forecast web service). This service is not under the control of a user application, or by a user's binding factory. That is, the web service has already been *exported.* The Fractal-BF supports application designers to establish distributed bindings also toward these services.

The Fractal-BF realizes these objectives by adopting a pure component-oriented approach: the acts of exporting and binding interfaces result in the creation of special components (*skeleton* and *stubs*, respectively), whose only purpose is to implement the distributed binding. This approach leverages the power of component-oriented technologies, permitting advanced control operations on such components.

A system architect can administer bindings. A number of operations are possible for an administrable binding: to monitor its undergoing activity (bandwidth consumption, average utilization, etc), to activate or de-activate it, or reconfigure it (statically or dynamically), etc.

The main goal of the Fractal-BF is to provide APIs and runtime support to establish and administer bindings.

5.2 Specification

This section details the two main primitives offered by the Fractal-BF to establish a (distributed) binding between two interfaces of remote components.

In the following of this section, the terms *client* and *server* interface of a Fractal component are used as in [7].

5.2.1 Export

The export operation requires as input: a reference to a Fractal component C, the name N of one of its server interfaces, a unique identifier P for the type of the binding to be established, and optional configuration parameters M to customise the default behaviour of P.
The result of this operation is the creation of a component K that can receives calls on behalf of C according to the communication protocol specificity's imposed by P and according to the M configuration parameters.
The K component is called *skeleton*.

The name of the K component is currently assigned by using the following pattern:

nameOf(C)-P-skeleton

Where:

1. *nameOf*(C) is a function that returns the name of the component C as given by the `org.objectweb.fractal.api.control.NameController` controller interface
2. P is the unique identifier of the binding type

Example: exporting an interface 'print' owned by a component with name 'printer' using a web service (P='ws') binding results in skeleton component K1, for which the following holds:

nameOf (K1) = printer-ws-skeleton

Skeleton components provide introspection capabilities by mean of the `org.objectweb.fractal.api.control.AttributeController (AC)` interface. Depending on the binding type being used, different implementations of AC are exposed by the skeletons.

InterfaceType of skeleton components
Skeleton component's interface types must be subtypes of the following type. The following ADL descriptor defines such interface type:

```
<definition name="org.objectweb.fractal.bf.SkeletonType" >
      <interface name="delegate"
            role="client"
            contingency="optional"

      signature="org.objectweb.fractal.api.Component" />
</definition>
```

During the export phase, the BF will establish a local binding between the skeleton's delegate client interface and the interface N of component C. This binding is a classical Fractal binding, and can be administered by system architects through the `org.objectweb.fractal.api.control.BindingController` interface. This local binding is required to let the skeleton dispatches incoming messages to the business interface N of component C.
The presence of the `delegate` client interface on the skeleton component is a requirement that could be removed in future versions of the Fractal-BF and completely rely on advanced bytecode generation techniques to create appropriate delegate interfaces. This is left for future versions of the project.

5.2.2 Bind

The bind operation takes as input a Fractal component C, the name N of one of its client interfaces, a unique identifier P for the type of the binding to be established, and optional configuration parameters M to customise the default behaviour of P.

The result of this operation is the creation of a component T that can dispatch method calls to remote services calls on behalf of C, according to the communication protocol specificity's imposed by P and the M configuration parameters.
The T component is called *stub*.
The name of the T component is defined by using the following pattern:

nameOf(C)-P-stub

Where:

1. *nameOf*(C) is a function that returns the name of the component C as given by the org.objectweb.fractal.api.control.NameController controller interface
2. P is the unique identifier of the binding type

Example: binding an interface 'printer' of a component 'printerClient' using a web service (P='ws') binding will result in the creation of a stub component T1, for which the following holds :

nameOf(T1)== printerClient-ws-stub

Stub components provide introspection capabilities by mean of the org.objectweb.fractal.api.control.AttributeController (AC) interface. Depending on the binding type being used, different implementations of AC are exposed by the stubs.

InterfaceType of stub components
Stub components interface types are generated using as input the interface type of the component C owning the interface N to be bound.

5.3 Architecture

This section details the component architecture of the Fractal Binding Factory. The Fractal-BF is itself a Fractal-based architecture. Its design is illustrated in the following picture.

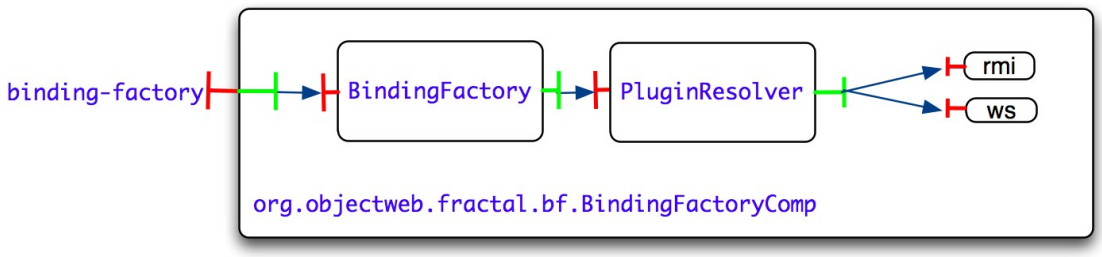


Figure 29: Fractal architecture of the Binding Factory

The main Binding Factory composite component (named org.objectweb.fractal.bf.BindingFactory-Comp) exposes one single server interface, named binding-factory, and doesn't required client interfaces.
The code for the principal interface is provided in Appendix (Section 12.2) of this report.

This main composite component includes, as a bare minimum configuration, two subcomponents, and optional components: a BindingFactory component primitive, a PluginResolver primitive component and, optionally, a set of components bound to the plugin resolver. This last set of components bound to the plugin resolver are also called *plugins*, and implement the real business logic to create stub and skeletons for a specific protocol. The internal structure of each plugin can be arbitrarily complex. A detailed description on the plugins and how to develop new ones can be found in Section 5.7 .

The reminder of this section is dedicated to the internal mechanisms currently implemented by the `org.objectweb.fractal.bf.BindingFactoryComp` component, and how the operations exposed by the `BindingFactory` interface are currently implemented.

5.3.1 BindingFactory

Goal

This is the primitive component which exposes the `org.objectweb.fractal.bf.BindingFactory` interface. Also, Its content provides the reference implementation for the `org.objectweb.fractal.bf.BindingFactory` interface. Therefore, it exposes a server interface of the same type of the composite component, and an internal binding between the two interfaces is established. Moreover, it requires a client interface toward a `PluginResolver`. Details about the scopes and the internal mechanisms of a `PluginResolver` are in the next section.

Usage

The following code shows how to use the BF through its Java APIs.

```
public class BFDemo {
  public static void main(String[] args) throws Exception{
    Factory adlFactory =
FactoryFactory.getFactory(FactoryFactory.FRACTAL_BACKEND);
    Component services = (Component)
adlFactory.newComponent("org.objectweb.fractal.bf.Services",new
HashMap());
    Component printer =
ContentControllerHelper.getSubComponentByName(services,"printer");
    Map<String, Object> exportHints = new HashMap<String, Object>();
    exportHints.put(BindingFactoryImpl.PLUGIN_ID, "ws");
    exportHints.put("address", "http://localhost:8080/Service");
    BindingFactory bindingFactory =
BindingFactoryHelper.getBindingFactory();
    bindingFactory.export(printer,"print",exportHints);
  }
}
```

This example exports the interface `print` of component `printer` using the `ws` plugin, by publishing a web-service available at a given address and port.

5.3.2 PluginResolver

Goal

The goal of the `PluginResolver` is to make available to the `BindingFactory` component the correct plugin specified by the user. To do so, a special <*key,element*> pair must be present in the export hints /bind hints specified by the user.
The key value *must* be `plugin.id`.
The currently supported values are: `ws`, `rmi`.
The current implementation of the `PluginResolver` goes through the following steps:

1. use the value of the `plugin.id` key to recover a reference to an already bound interface to its collection interface `plugins` with the composite name `plugins-(plugin.id)`
2. if one is found, a reference to it is returned
3. if none is found, the following steps are executed:

A. a Fractal ADL definition name is created, using the following schema:

```
"org.objectweb.fractal.bf.connectors."+valueOf(plugin.id+"."capitalize(valueOf(plu
gin.id))+"Connector"
```

For instance, given a `plugin.id` value of '`rmi`', the following Fractal ADL definition name is built: `org.objectweb.fractal.bf.connectors.rmi.RmiConnector`.
The current implementation of the PluginResolver doesn't provide a mean to customise such naming conventions; future versions might provide such support;

B. a component with the specified name is loaded

C. the server interface '`plugin`' of this plugin component is bound to the collection interface `plugins` of the `PluginResolver`, with the name: `plugins-(plugin.id)`

5.4 Core operations

This section details how the core operations provided by the BF are implemented in the reference implementation. The focus is on the *export* and *bind* operations, leaving the specifications of the unexport and unbind operations to future releases of this document.

5.4.1 Export

3) the BF uses the input parameters to locate the server interface of the component C to export;
4) the BF uses the `plugin.id` parameter to retrieve the plugin P to be used;
5) P is used to create a skeleton S component
6) the skeleton component is bound to the Component interface of the C component
7) the skeleton component is started.

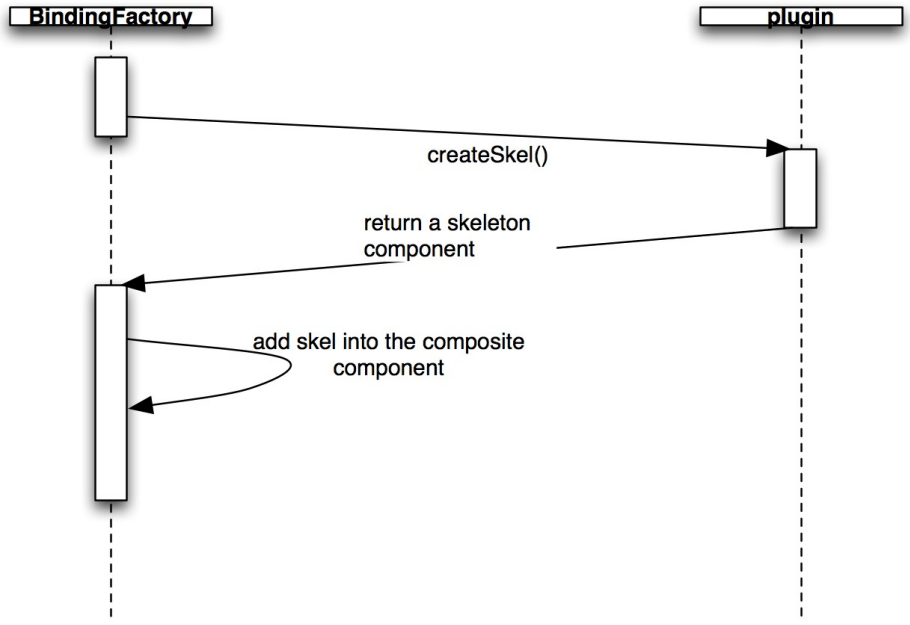


Figure 30: Sequence diagram for the export

Figures 31 and 32 illustrates the process of exporting a `print` interface owned by a component `printer` using a web-service (`ws`) plugin.

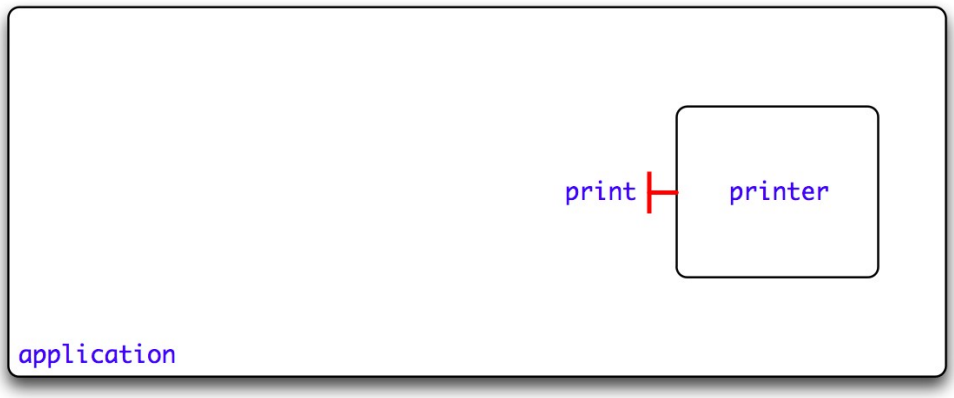


Figure 31: Before exporting the print interface using the web service plugin

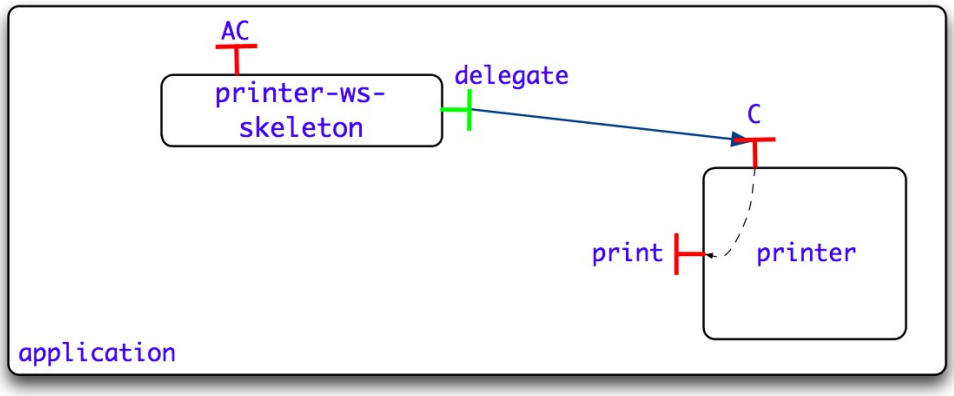


Figure 32: After exporting the print interface using the web service plugin

Note that the internal binding between the `org.objectweb.fractal.api.Component` interface `C` of the `printer` component and its public `print` server interface is currently required to dispatch method calls received by the skeleton component to the business component. Such requirement could not apply on future releases of the BF.
The `AC` controller interface of the `printer-ws-skeleton` allows for introspection operations on the skeleton component.

5.4.2 Bind

1) the BF uses the input parameters to locate the client interface of the component C to bind;
2) the BF uses the `plugin.id` parameter to retrieve the plugin P to be used;
3) P is used to create a stub S component
4) S is bound , through a server interface it exposes, to the client interface of the client component C;
5) S is started

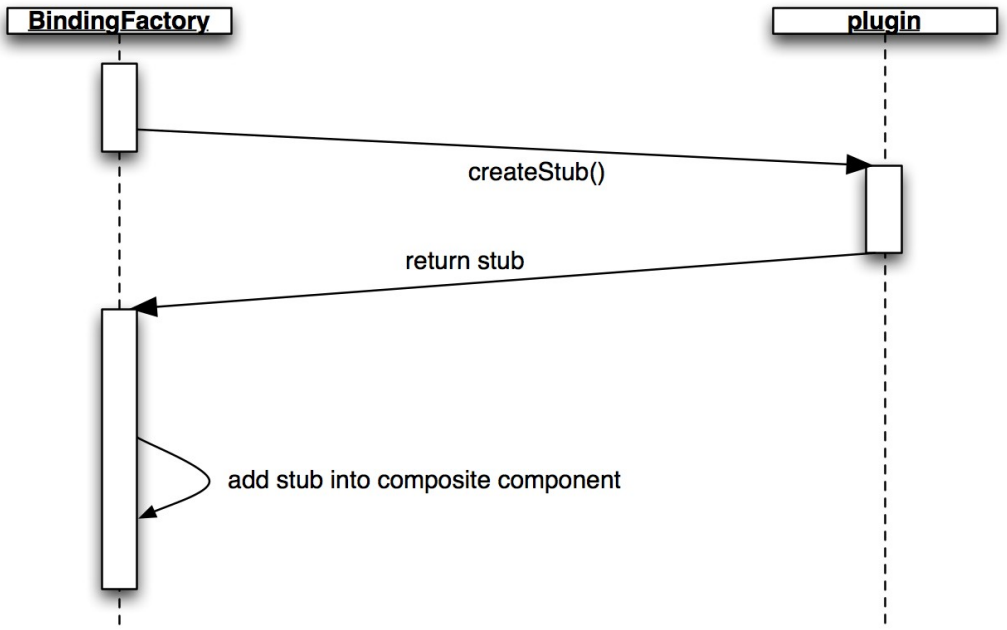


Figure 33: Sequence diagram of the steps executed during the bind

Figures 34 and 35 illustrate how the internal architecture of an application evolves during the process of binding a `print` interface owned by a component `printerClient` using a web-service (`ws`) plugin. The `AC` controller interface of the `printerClient-ws-stub` allows for introspection operations on the stub component.

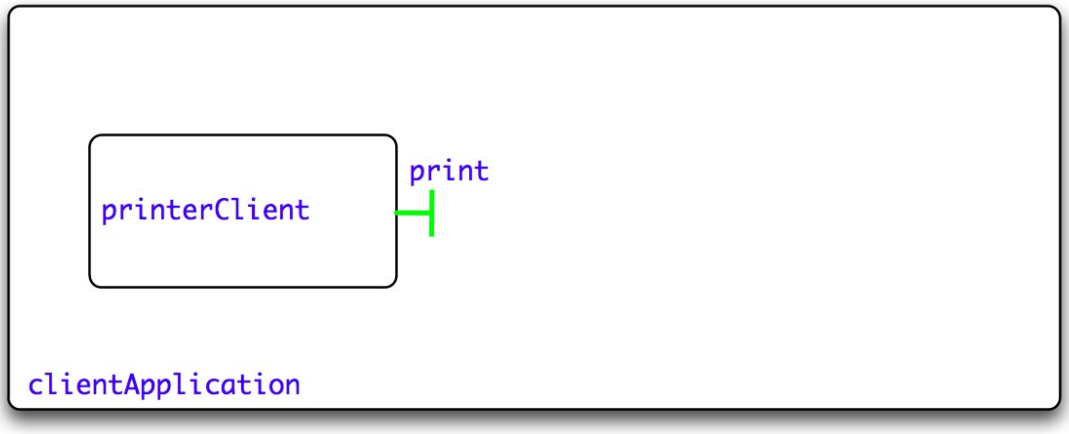


Figure 34: Before binding the print interface using the web service plugin

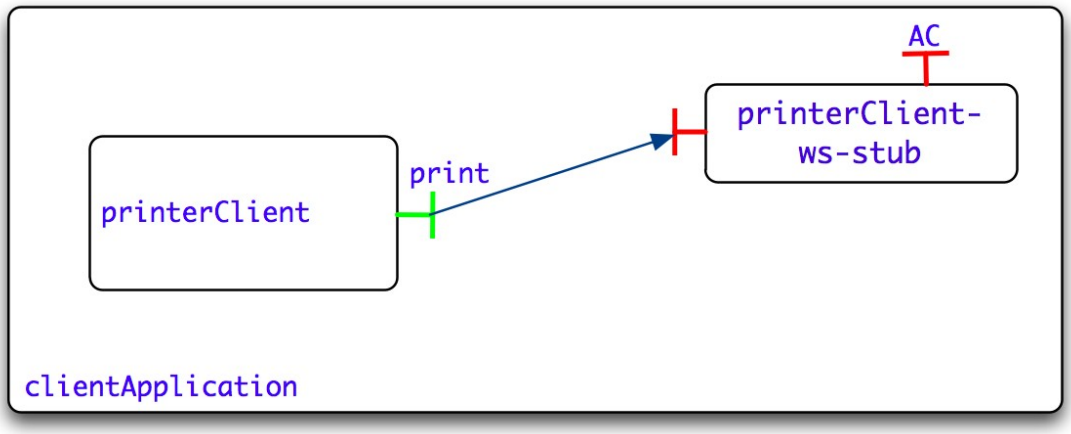


Figure 35: After binding the print interface using the web service plugin

5.5 Binding Factory and SCA Applications

The BF is currently embedded in the FraSCAti runtime, as the component in charge of creating wires between SCA references and SCA services.

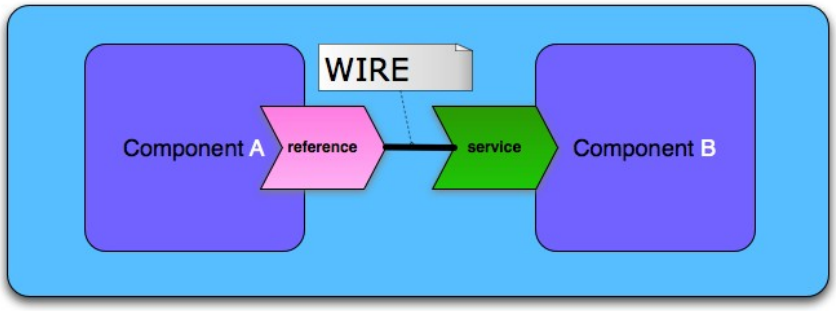


Figure 36: Wire construction in SCA applications

5.6 Plugins API

The development of new plugins to support other binding protocols revolves around providing a Fractal component that can be then bound to the `PluginResolver` component.

To be a valid plugin, this Fractal component must provide a server interface named `plugin` and whose signature is a subtype of `org.objectweb.fractal.bf.BindingFactoryPlugin<E extends ExportHints, B extends BindHints>`. Details about this interface can be found in Appendix A.
Each plugin *must* define two concrete implementations of the `ExportHints` and `BindHints` interface (provided by the Fractal BF API). Figure 37 shows the external architecture of a BF plugin. The internal architecture of a plugin component can be arbitrarily complex.

Figure 37: External design of a Binding Factory plugin

5.7 Fractal ADL support

It is possible to use the Binding Factory with FractalADL. Here's an example:

```
01 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
   EN" "classpath://org/objectweb/fractal/bf/adl/xml/standard-bf.dtd">
02 <definition name="org.objectweb.fractal.bf.Services">
03   <interface name="print" role="server"
       signature="org.objectweb.fractal.bf.Print" />
04   <component name="service">
05     <interface name="print" role="server"
06       signature="org.objectweb.fractal.bf.Print" />
07     <content class="org.objectweb.fractal.bf.PrintImpl" />
08   </component>
09   <binding client="this.print" server="service.print" />
10   <exporter type="ws" interface="service.print">
11     <parameter name="address" value="http://localhost:8080/Print" />
12   </exporter>
13 </definition>
```

Line 01 declares the DTD used in this ADL file (standard-bf.dtd): it is required to use the BF extensions. This DTD is an extension to the FractalADL standard.dtd, and it adds two new elements (`exporter` and `binder` elements). This DTD file becomes available only after having imported the `fractal-bf-adl` Maven module.

Line 02 defines a composite component named '`org.objectweb.fractal.bf.Services`'.

Line 03 defines an external '`print`' server interface, whose Java type is '`org.objectweb.fractal.bf.Print`'.

Lines 04-08 define a '`service`' component with a server interface '`print`'.

Line 09 defines an internal binding between the external '`print`' server interface and the '`print`' server interface of the '`service`' component.

Lines 10-12 define an '`exporter`' element: two required attributes must be provided:

1. type: a unique string identifier, used by the BF plugin resolver to load the appropriate plugin;
2. interface: a '{component}.{interfaceName}' identifier for the interface to export; in this case, the 'service.print' identifies the 'print' interface declared by the 'service' component

Line 11 defines an optional '`parameter`' element: plugins might declare any number of `<parameter>` elements.
Each parameter element requires two attributes:
- `name` : the name of the attribute
- `value`: the value for the attribute
The parameters accepted by a plugin are plugin-dependant. New plugins can support arbitrarily chosen parameters.

5.8 Implementation

The current implementation of the Binding Factory is organized as a Maven 2 project. The modules comprehend the core modules, a set of plugins, integration tests and examples.
The sources of the BF were first committed in the official Fractal SVN repository the 20th May, 2008, in its early 0.1 version. Lately, it's been moved in the Fractal trunk repository since the 13th November, 2008.
The latest stable version (0.5) has been deployed on 17th February, 2008. A 0.6 version is currently under development.
The sources can be explored by browsing the address:
http://svn.forge.objectweb.org/cgi-bin/viewcvs.cgi/fractal/trunk/fractal-bf/
The binary packages are published on the official OW2 maven repository, and they can be easily imported into a Maven2-based project by including the following dependencies to the pom.xml file:

```
<properties>
      <bf.version>0.5</bf.version>
</properties>
<dependency>
      <groupId>org.objectweb.fractal.bf</groupId>
      <artifactId>fractal-bf-core</artifactId>
      <version>${bf.version}</version>
</dependency>
<dependency>
      <groupId> org.objectweb.fractal.bf </groupId>
      <artifactId>fractal-bf-adl</artifactId>
      <version>${bf.version}</version>
</dependency>
<!-- only if you need WebService bindings-->
<dependency>
      <groupId>org.objectweb.fractal.bf.connectors</groupId>
      <artifactId>fractal-bf-connectors-soap-cxf</artifactId>
      <version>${bf.version}</version>
</dependency>
```

To resolve correctly these Maven dependencies (either for stable or snapshot releases), the following repositories must be added:

```
<repositories>
            <repository>
                  <id>objectweb-release</id>
                  <name>ObjectWeb Maven Repository</name>
                  <url>http://maven.objectweb.org/maven2</url>
                  <releases>
                        <enabled>true</enabled>
                  </releases>
                  <snapshots>
                        <enabled>false</enabled>
                  </snapshots>
            </repository>
            <repository>
                  <id>objectweb-snapshot</id>
                  <name>ObjectWeb Maven Repository</name>
                  <url>http://maven.objectweb.org/maven2-snapshot</url>
                  <releases>
                        <enabled>false</enabled>
                  </releases>
                  <snapshots>
                        <enabled>true</enabled>
                        <updatePolicy>daily</updatePolicy>
                  </snapshots>
            </repository>
      </repositories>
```

6 Transaction Service

This chapter contains specifications related to the Task 1.5 of the SCOrWare project. This is dedicated to the design and implementation of the SCOrWare transaction service. The objectives of the SCOrWare transaction service are outlined in Section 3.1. The architecture, the specification, and implementation of the SCOrWare transaction service are discussed in Section 6.2, 6.3, and 6.4 respectively.

6.1 Objectives

Transactions in a software architecture are non-functional properties. The objectives of a transaction service are to guarantee that a business code of an SCA component (or composite), or more generally an SCA domain, observes ACID properties (Atomicity, Consistency, Isolation, Durability).

6.2 Architecture

In FraSCAti, transactions are managed via SCA Intents. A unique Transaction Manager is in charge of supporting transactions.
Transaction intents are weaved to SCA components via the Tinfi APIs, especially with the SCAIntentController and the SCAIntentHandler interfaces.

The architecture of the transaction manager is described as an SCA composite.

Figure 38 illustrates how transaction intents are wired to the transaction manager in order to adapt the policies to ensure the support for local transactions.

As shown in figure 38, the transaction intent manager composite gathers all transaction intents. Then, using a given intent only means to bind the Intent Controller of Tinfi to the corresponding service that promotes the intent service of the required intent.

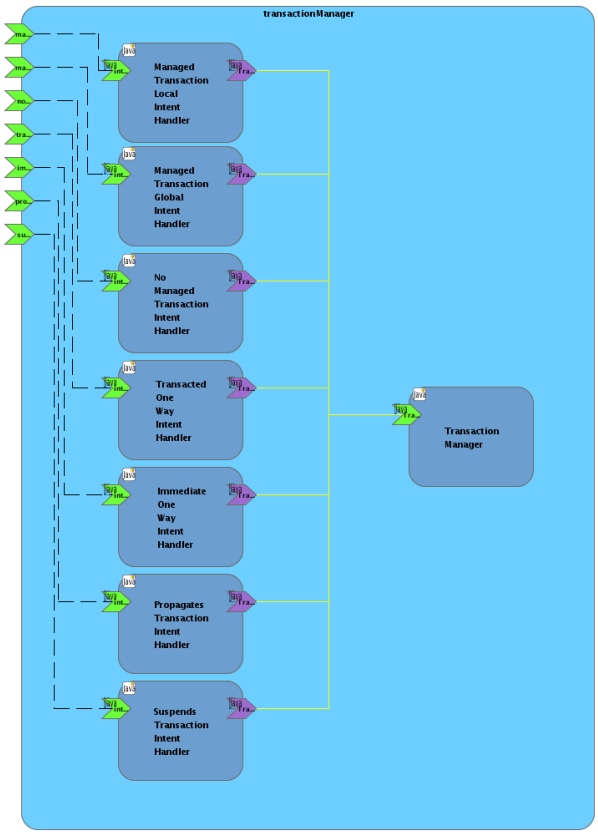


Figure 38: Architecture of the Transaction Intent Manager

6.3 Specification

The FraSCAti transaction service composes the JTA specification (http://java.sun.com/javaee/technologies/jta/index.jsp) and the SCA Transaction Policies (http://www.osoa.org/download/attachments/35/SCA_TransactionPolicy_V051b.pdf).

6.3.1 Java Transaction API

The FraSCAti platform provides a unique Transaction Manager per domain. It allows to perform local transactions within a set of SCA colocated components (composites).
FraSCAti does not support JTS specification (distributed transactions). As a consequence, runtime applications are responsible for ensuring transaction context demarcation and transactions propagation.
FraSCAti ensures support of local transactions through SCA policies, implemented as SCA intents.

6.3.2 SCA Transaction Policies

The transaction policies enable the capability for SCA components to perform business logic in a transacted context.
Each transaction policy corresponds to a given SCA composite, with the same name. Available policies are:

- managedTransaction.local
- managedTransaction.global
- noManagedTransaction
- propagatesTransaction
- suspendsTransaction
- transactedOneWay
- immediateOneWay

6.4 Implementation

The implementation framework of the FraSCAti transaction service is JOTM (http://jotm.ow2.org).
The FraSCAti Transaction Manager component delegates the transaction management to the JOTM Transaction Factory.
According to a given transaction policy, the corresponding intent handler will adapt the delegation to the JOTM transaction engine.

6.4.1 Managed and non-managed transactions

- managedTransaction.global:
  There must be an atomic transaction in order to run this component. FraSCAti must ensure that a global transaction is present before dispatching any method on the component. It uses any transaction propagated from the client or else begins and completes a new transaction.
  When using this intent, FraSCAti will begin a transaction before running the business logic of the component. When done, the transaction is committed if no business error occurs, else, the transaction is rolled-back.
- managedTransaction.local
  The component cannot tolerate running as part of a global transaction, and will therefore run within a local transaction containment (LTC) that is started and ended by the SCA runtime. Any global transaction context that is propagated to the hosting SCA runtime must not be visible to the target component. Any interaction under this policy with a resource manager is performed in an extended resource manager local transaction (RMLT). Upon successful completion of the invoked service method, any RMLTs are implicitly requested to commit by the SCA runtime. Note that, unlike the resources in a global transaction, RMLTs so coordinated in a LTC may fail independently. If the invoked service method completes with a non-business exception then any RMLTs are implicitly rolled back by the SCA runtime. In this context a business exception is any exception that is declared on the component interface and is therefore anticipated by the component implementation. The manner in which exceptions are declared on component interfaces is specific to the interface type– for example Java interface types declare Java exceptions, WSDL interface types define wsdl:faults. Local transactions cannot be propagated outbound across remotable interfaces.

When using this intent, a local transaction context is created for the execution of the component. That means: if there is no active transaction, the behaviour of this intent is the same as managedTransaction.global, and if there is an active transaction, this transaction is suspended, a new one is created before the business logic of the component and committed or rolled back according to errors detection. Then after this, the suspended transaction is resumed.

- noManagedTransaction
  The component runs without a managed transaction, under neither a global transaction nor an LTC. A transaction that is propagated to the hosting SCA runtime MUST NOT be joined by the hosting runtime on behalf of this component. When interacting with a resource manager under this policy, the application (and not the SCA runtime) is responsible for controlling any resource manager local transaction boundaries, using resource-provider specific interfaces (for example a Java implementation accessing a JDBC provider must choose whether a Connection should be set to autoCommit(true) or else must call the Connection commit or rollback method). SCA defines no APIs for interacting with resource managers.
  When using this intent, FraSCAti will check that, before running the component, there is no active transaction currently active in the domain. If there is one, an error is thrown and the business logic of the component is not proceeded.

6.4.2 OneWay invocations

- transactedOneWay
  When applied to a reference indicates that any OneWay invocation messages MUST be transacted as part of a client global transaction. If the client is not configured to run under a global transaction or if the binding does not support transactional message sending, then a deployment error occurs. When applied to a service indicates that any OneWay invocation message MUST be received from the transport binding in a transacted fashion, under the target service's global transaction. The receipt of the message from the binding is not committed until the service transaction commits; if the service transaction is rolled back the the message remains available for receipt under a different service transaction. If the service is not configured to run under a global transaction or if the binding does not support transactional message receipt, then a deployment error occurs.
- immediateOneWay
  When applied to a reference indicates that any OneWay invocation messages is sent immediately regardless of any client transaction. When applied to a service indicates that any OneWay invocation is received immediately regardless of any target service transaction. The outcome of any transaction under which an immediateOneWay message is processed has no effect on the processing (sending or receipt) of that message.

6.4.3 Transaction interaction policies

- propagatesTransaction
  The SCA runtime MUST ensure that the service is dispatched under any propagated (client) transaction. Use of the propagatesTransaction intent implies that the service binding MUST be capable of receiving a transaction context and that a service with this intent specified will always join a propagated transaction, if present. However, it is important to understand that some binding/policySet combinations that provide this intent for a service will require the client to propagate a transaction context. In SCA terms, for a reference wired to such a service, this implies that the reference must use either the propagatesTransaction intent or a binding/policySet combination that does propagate a transaction. If, on the other hand, the service does not require the client to provide a transaction (even though it has the capability of joining the client's transaction), then some care is needed in the configuration of the service. One approach to consider in this case is to use two distinct bindings on the service, one that uses the propagatesTransaction intent and one that does not - clients that do not propagate a transaction would then wire to the service using the binding without the propagatesTransaction intent specified.
- suspendsTransaction
  The SCA runtime MUST ensure that the service is NOT dispatched under any propagated (client) transaction.
  When using this intent, if there is an active transaction when running business logic of the component, FraSCAti will suspend the current transaction, run the component, then resume the suspended transaction.

7 Trading and Composer Services

This chapter contains specifications related to the Task 1.6 of the SCOrWare project. This is dedicated to the design and implementation of the semantic trading and composing services. Section 7.1 describes the global trading and composing service architecture. The Section 7.2 describes the semantic trading system and Section 7.3 the composer.

7.1 The semantic system

7.1.1 Trading and Composer Services Architecture

Trading is a technique, which consists in giving to a service component the possibility of dynamically discovering the components adapted to its needs (references). By Composing we meant the action of transforming the description of a composite, by replacing abstract subcomponents with concrete ones, so that the composite description, from abstract, becomes shallow or deep concrete. Six entities intervene in these processes: the service provider, the service consumer, the administrator, the Registry , the SemanticTrader and the Composer (see the figure 39).

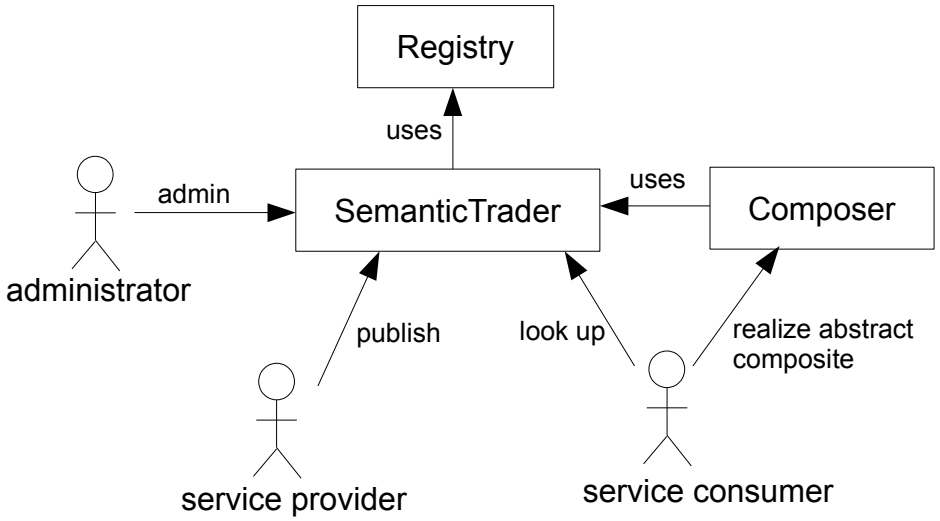


Figure 39: Trading Service

The semantic trader and composer play the role of intermediary between the other entities and thus allows them to interact in a distributed environment. The provider is the entity which uses the trader to publish information describing the services that it offers. The consumer is the entity which calls upon the trader to search services in the registry, or calls upon the composer to turn a composite from abstract to concrete. For the trader, even an incomplete description of the required component can be used in search. The registry is the entity used to store information describing the offered services, the comsumers and the providers. The administrator manages mainly the rights of access to the trading service (publishing and searching rights).

7.1.2 Semantic Description of SCDL Specifications

To support a semantic trading, we propose SASCDL (Semantic Annotation for SCDL), a semantic approach for annotation of SCDL descriptions inspired by the semantic annotation of the Web Services descriptions (i.e. SAWSDL). This choice will allow an incremental semantic description, the discovery and the dynamic invocation of SCA components and an independence of the semantic model representation languages.
SASCDL does not specify a language for the representation of the semantic models. It provides mechanisms to reference the concepts of a semantic model defined outside SCDL document by adding annotations to the elements in the SCDL document. For this purpose, we use two extensions: *modelReference* and *schemaMapping*. This is carried out by using the “sawsdl” attribute followed by one of these extensions. *modelReference* is a set of URIs separated by

spaces and associated with the elements of a SCDL description. For example :

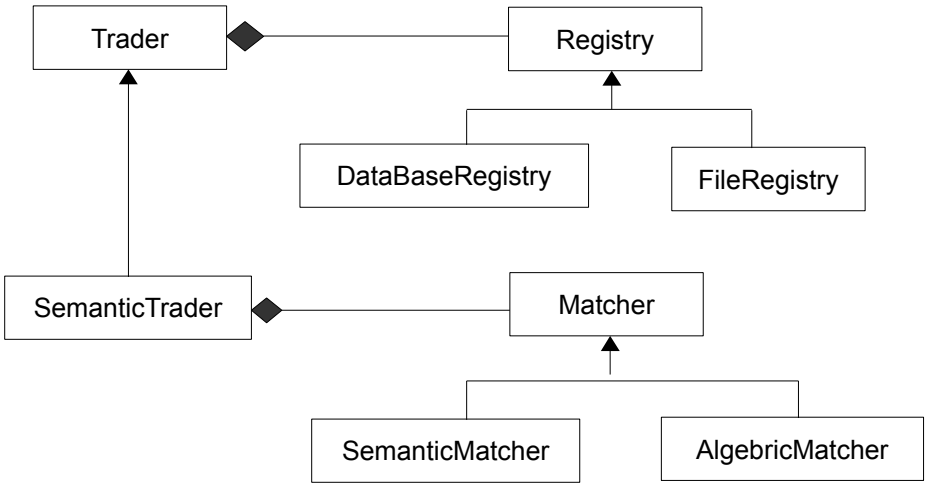
```
<sca:component name="CarRentalComponent"
  sawsdl:modelReference="file:resources/ontologies/rental.owl#CarRental">
      <sca:implementation.composite name=”BigCarRental”/>
      <sca:service name="CarRentalService”
        sawsdl:modelReference="file:resources/ontologies/rental.owl#CarRental"/>
</sca:component>
```

The *schemaMapping* is used to carry out the correspondence between the data structures described in a SCDL specification. *schemaMapping* is useful when the XML structures required by the consumer and those provided by the provider are different, thus their translation in a semantic model solves the discrepancy.

7.2 The semantic trading

7.2.1 The SemanticTrader architecture

The SemanticTrader returns for a given abstract component description, the description of a component semantically equivalent but concrete. In order to do so it specializes the Trader which provides an extensive access to the Registry that contains the concrete component descriptions. The SemanticTrader also holds a Matcher with which it can compare the abstract component given as argument with the ones returned by the registry.



The Registry can have various implementations depending on the way the available concrete components are serialized. Currently we provide a database implementation. The DatabaseRegistry uses a mySQL base in which components are stored in a simple table that contains for each component : its name, its xml description, its unique key determined on its registration request, its provider id. We also plan to provide a FileRegistry in which composite are serialized in files. The Matcher has also two main implementations. The SemanticMatcher compares two components on the base of semantical annotations, and the AlgebricMatcher is implemented by the IRIT Partner.

7.2.2 The SemanticTrader interfaces specification

We distinguish four interfaces for the use of the trading service: the TraderAdministrationInterface interface, the TraderPublicationInterface interface, the TraderSearchInterface interface and the SemanticTraderSearchInterface interface. Only the SemanticTraderSearchInterface interface appears in the SemanticTrader and not in the Trader.

a) TraderAdministrationInterface Interface

The TraderAdministrationInterface interface allows the administrator of the trading service to manage the users' rights of the trading service (customer and provider components). Users are identified through their "Id". Each user owns a key with which are associated its rights, and that he/she must provide for publishing/requesting components.

```
String saveUser (User user, String adminKey)
```

The saveProvider operation enables the administrator of the trading service to add in the directory a provider or a customer of components. It takes as arguments a user instance (*user*) and the key of the admin (*adminkey*), then returns the key of the user if added or empty String otherwise. Only the suppliers added by the administrator will have the right to publish components in the directory. Only the customers added by the administrator will have the right to use the trading service to ask the directory.

```
boolean deleteUser (String userId, String adminKey)
```

The deleteUser operation enables the administrator of the trading service to remove a user from the directory. It takes as arguments the id of the user (*userId*) and the key of the administrator (*adminKey*), then returns true if success and false otherwise. In case the user is a provider, then all its registered components are also removed.

```
String getUserKey (String userId, String adminKey)
```

This method returns the key of a user. It takes as arguments the id of the user (*userId*) and the key of the administrator (*adminKey*).

```
String[] getProvidersId (String requesterKey)
```

This method returns the list of the providers id. It takes as argument the key of the requester (*requesterKey*).

```
String[] getCustomersId(String requesterKey)
```

This method returns the list of the customers id. It takes as argument the key of the requester (*requesterKey*).

b) TraderPublicationInterface Interface

The TraderPublicationInterface interface allows the providers of authorized services to publish, or remove information concerning the services they offer. This interface offers the operations *saveComponent*, *saveComposite*, *saveComponents*, *deleteComponent*, and *deleteComposite*.

The *saveComponent* operation allows the authorized provider of a service to publish a component in the registry. The *saveComposite* operation allows authorized service providers to publish a composite in the registry. The *saveComponents* operation allows authorized service providers to publish all the components within a composite in the registry. The *deleteComponent* operation allows removing a component from the registry by an authorized provider. The *deleteAbstractComposite* and *deleteConcreteComposite* allows removing a composite from the registry by an authorized provider.

```
String saveComponent (Component component, Provider provider)
```

The saveComponent operation enables an authorized provider of a service to publish a component in the directory. It takes as arguments the description instance of the component to be published (*component*) and the provider instance (*provider*), and returns the key of the saved component if success and null otherwise.

```
String saveComposite (Composite composite, Provider provider)
```

The saveComposite operation enables an authorized provider of a service to publish a composite in the directory. It takes as arguments the description instance of the composite to be published (*composite*) and the provider instance (*provider*), and returns the key of the saved composite if success and null otherwise.

```
String saveComponent (Composite composite, Provider provider)
```

This method saves the unique concrete component within a composite. It takes as arguments the description instance of the composite containing the component to be published (*composite*) and the provider instance (*provider*), and returns the key of the saved component if success and null otherwise.

```
List<String> saveComponents (Composite composite, Provider provider)
```

The saveComponents operation enables an authorized provider of a service to publish all the components within a composite in the directory. It takes as arguments the description instance of the composite containing the components to be published (*composite*) and the provider instance (*provider*), and returns the keys of the saved components if success and null otherwise.

```
String saveComposite (String compositePath, Provider provider)
```

This method enables an authorized provider of a service to publish a serialized composite in the directory. It takes as arguments the composite file path of the composite to be published (*compositePath*) and the provider instance (*provider*), and returns the key of the saved composite if success and null otherwise.

```
String saveComponent (String compositePath, Provider provider)
```

This method saves the unique concrete component within a serialized composite. It takes as arguments the composite file path of the composite containing the component to be published (compositePath) and the provider instance (provider), and returns the key of the saved component if success and null otherwise.

```
boolean deleteComponent (String componentKey, String providerKey)
```

The deleteComponent operation enables an authorized provider of a service to remove one of its components from the directory. It takes as arguments the key of the component (*componentKey*) and the key of the provider (*providerKey*), then returns true if the component is removed from the registry.

```
boolean deleteAbstractComposite (String compositeKey, String providerKey)
```

The deleteAbstractComposite operation enables an authorized provider of a service to remove one of its abstract composites from the directory. It takes as arguments the key of the composite (*compositeKey*) and the key of the provider (*providerKey*), then returns true if the composite is removed from the registry.

```
boolean deleteConcreteComposite (String compositeKey, String providerKey)
```

The deleteConcreteComposite operation enables an authorized provider of a service to remove one of its concrete composites, given by its key, from the directory. It takes as arguments the key of the composite (*compositeKey*) and the key of the provider (*providerKey*), then returns true if the composite is removed from the registry.

c) *TraderSearchInterface Interface*

The TraderSearchInterface interface allows the authorized consumers of services to search the offered SCA components and SCA composites registered by providers. This interface offers operations:
–that find components by name, provider, key
–that find abstract composites by name, provider, key
–that find concrete composites by name, provider, key
The CompInfo object used by these methods gives access, for the component with which it is initialized, to :
–the EObject that models the component,

–the XML text describing the component,

```
List<CompInfo> findComponents (String customerKey)
```

This method returns all the components registered in the directory. It takes as argument the key of the requester (*customerKey*) and returns a list of CompInfo related to the searched components.

```
List<CompInfo> findComponentsOfProvider (String providerId, String customerKey)
```

This method returns all the components published by a provider. It takes as arguments the key of the requester (*customerKey*), the id of the provider of requested components (*providerId*) and returns a list of CompInfo related to the searched components.

```
List<CompInfo> findComponentsWithName (String componentName, String customerKey)
```

This method returns all the components whose name is given. It takes as arguments the key of the requester (*customerKey*), the name of the requested components (*componentName*) and returns a list of CompInfo related to the searched components.

```
List<CompInfo> findComponentsOfProviderWithName (String componentName,
  String providerId, String customerKey)
```

This method returns all the components published by a given provider and whose name is given. It takes as arguments the key of the requester (*customerKey*), the name of the requested components (*componentName*), the id of the provider of requested components (*providerId*) and returns a list of CompInfo related to the searched components.

```
CompInfo findComponentWithKey (String componentKey, String customerKey)
```

This method returns the unique component whose key is given. It takes as arguments the key of the requester (*customerKey*), the key of the requested components (*componentKey*) and returns the CompInfo related to the searched component.

```
List<CompInfo> findAbstractComposites (String customerKey)
```

This method returns all the abstract composites registered in the directory. It takes as argument the key of the requester (*customerKey*), and returns a list of CompInfo related to the searched components.

```
List<CompInfo> findAbstractCompositesOfProvider (String providerId,
  String customerKey)
```

This method returns all the abstract composites published by a given provider. It takes as arguments the key of the requester (*customerKey*), the id of the provider (*providerId*) and returns a list of CompInfo related to the searched components.

```
List<CompInfo> findAbstractCompositesWithName (String compositeName,
  String customerKey)
```

This method returns all the abstract composites whose name is given. It takes as arguments the key of the requester (*customerKey*), the name of the composite (*compositeName*) and returns a list of CompInfo related to the searched components.

```
List<CompInfo> findAbstractCompositesOfProviderWithName (String compositeName,
  String providerId, String customerKey)
```

This method returns all the abstract composites published by a given provider and whose name is given. It takes as arguments the key of the requester (*customerKey*), the name of the composite (*compositeName*), the id of the provider (*providerId*) and returns a list of CompInfo related to the searched components.

```
CompInfo findAbstractCompositeWithKey (String compositeKey, String customerKey)
```

This method returns the unique abstract composite whose key is given. It takes as arguments the key of the requester (*customerKey*), the key of the composite (compositeKey), and returns the CompInfo related to the searched component.

```
List<CompInfo> findConcreteComposites (String customerKey)
```

This method returns all the registered concrete composites. It takes as argument the key of the requester (*customerKey*), and returns a list of CompInfo related to the searched components.

```
List<CompInfo> findConcreteCompositesOfProvider (String providerId,
  String customerKey)
```

This method returns all the concrete composites published by a given provider. It takes as arguments the key of the requester (*customerKey*), the id of the provider (*providerId*) and returns a list of CompInfo related to the searched components.

```
List<CompInfo> findConcreteCompositesWithName (String compositeName,
  String customerKey)
```

This method returns all the concrete composites whose name is given. It takes as arguments the key of the requester (*customerKey*), the name of the composite (*compositeName*) and returns a list of CompInfo related to the searched components.

```
List<CompInfo> findConcreteCompositesOfProviderWithName (String compositeName,
  String providerId, String customerKey)
```

This method returns all the concrete composites published by a given provider and whose name is given. It takes as arguments the key of the requester (*customerKey*), the name of the composite (*compositeName*), the id of the provider (*providerId*) and returns a list of CompInfo related to the searched components.

```
CompInfo findConcreteCompositeWithKey (String compositeKey, String customerKey)
```

This method returns the unique concrete composite whose key is given. It takes as arguments the key of the requester (*customerKey*), the key of the composite (compositeKey), and returns the CompInfo related to the searched component.

d) SemanticTraderSearchInterface Interface

This interface provides methods that return, from the registry, concrete components semantically equivalent to the given abstract ones.

```
List<CompInfo> findConcreteComponents (Component abstractComponent,
  CustomerKey customerKey)throws NullCustomerKeyException
```

This method returns all the concrete components that matches with the abstract one. It takes as arguments the key of the requester (*customerKey*), the abstract component description instance (*abstractComponent*), and returns a list of CompInfo related to the matching concrete components. It will raise a *NullCustomerKeyException* if the customer key is null.

```
List<CompInfo> findConcreteComponents (Component abstractComponent,
  String providerId, CustomerKey customerKey) throws NullCustomerKeyException
```

This method returns all the concrete components published by a provider that matches with the abstract one. It

takes as arguments the key of the requester (*customerKey*), the abstract component description instance (*abstractComponent*), the id of the provider (*providerId*), and returns a list of CompInfo related to the matching concrete components. It will raise a *NullCustomerKeyException* if the customer key is null.

```
List<CompInfo> findConcreteComposites (Composite abstractComposite,
  CustomerKey customerKey) throws NullCustomerKeyException
```

This method returns all the concrete composites that matches with the abstract one. It takes as arguments the key of the requester (*customerKey*), the abstract composite description instance (*abstractComposite*), and returns a list of CompInfo related to the matching concrete composites. It will raise a *NullCustomerKeyException* if the customer key is null.

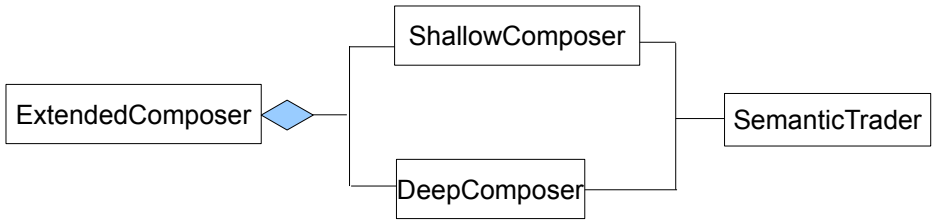
```
List<CompInfo> findConcreteComposites (Composite abstractComposite,
  String providerId, CustomerKey customerKey) throws NullCustomerKeyException
```

This method returns all the concrete composites published by a provider that matches with the abstract one. It takes as arguments the key of the requester (*customerKey*), the abstract composite description instance (*abstractComponent*), the id of the provider (*providerId*), and returns a list of CompInfo related to the matching concrete composites. It will raise a *NullCustomerKeyException* if the customer key is null.

7.3 The ExtendedComposer

7.3.1 The ExtendedComposer architecture

The ExtendedComposer is the class in charge of the transformation of the abstract composition description. It is a "Facade" for the ShallowComposer and the DeepComposer which are respectively in charge of building shallow and deep concrete composites from abstract ones. They both rely on the SemanticTrader to retrieve concrete components before including them in the abstract composite in a way that can be parameterized.



7.3.2 The ExtendedComposer interface specification

The ExtendedComposer provides two main kinds of transformation of composite:
- the shallow transformation that makes composites shallow concrete by changing only direct children abstract subcomponents for concrete ones : through the **shallowAbstractToConcrete** methods,
- the deep transformation that makes composites deep concrete by changing all abstract subcomponents until leaves for concrete ones : through the **deepAbstractToConcrete** methods,

The ExtendedComposer is able to work either on composite files and in memory composites. The ModificationHistory object summarizes all the modifications made to a composite.

Composite **shallowAbstractToConcrete** (Composite abstractComposite)

This method builds a composite description, shallow concrete, semantically equivalent to the abstractComposite argument, by modifying its direct children. It takes as arguments the description instance of the composite that is shallow abstract (*abstractComposite*) and returns a composite description, semantically equivalent to the abstractComposite argument.

```
void shallowAbstractToConcrete (String abstractFile, String concreteFile)
```

This methods reads a composite description file, builds the description, makes it shallow concrete, and writes it to another file. It takes as arguments the original composite short file name (*abstractFile*) and returns the file in which write the result of the transformation (*concreteFile*).

```
ModificationHistory getShallowModificationHistory ()
```

This method returns the ModificationHistory of the last shallow transformation.

```
void deepAbstractToConcrete (String rootCompositeFileName,
  String sourceDirectory, String destinationDirectory)
```

This method rewrites a composite description and all it associated files in order to make the composite deep concrete. It takes as arguments the short file name of the composite description file (*rootCompositeFileName*), the directory containing files (for example : composite implementations of subcomponents, or “includes”...) referenced by the composite description (*sourceDirectory*), the directory in which composite files are written (*destinationDirectory*).

```
void deepAbstractToConcrete (Composite rootComposite, String sourceDirectory)
```

This method modifies a composite description object in order to make it deep concrete. It takes as arguments the composite description object (*rootComposite*), and the directory containing the files (for example : composite implementations of subcomponents, or “includes”...) referenced by the composite description (*sourceDirectory*).

```
ModificationHistory getDeepModificationHistory ()
```

This method returns the ModificationHistory of the last deep transformation.

7.4 Summary

In this section, we have focused on the architecture of a semantic trading and composing services. In this architecture we have mainly identified relationships and interfaces between the administrator, the service provider, the service consumer, and the registry, the semantic trader and the extended composer. We have, in addition, proposed a semantic annotation for SCDL descriptions that provides mechanisms to reference concepts of external semantic models. This allows us to be independent of the representation language of the used semantic model. Finally, we have specified the different operations to be provided by the semantic trading and composing services.

8 Deployment and Autonomic Support

This chapter contains specifications related to the Task 1.7 of the SCOrWare project. This is dedicated to the deployment and autonomic support of SCA-based applications and runtime environments. Section 8.1 discusses the FraSCAti assembly factory in charge of parsing and validating SCA descriptors, then instantiating SCA composites, components and bindings. Section 8.2 addresses the deployment of SCA-based applications and runtime environments thank to the Fractal Deployment Framework (FDF). Section 8.3 presents the autonomic support of SCA-based systems with TUNe.

8.1 SCA Assembly Factory

8.1.1 Objectives

The SCA assembly factory is part of the SCOrWare platform (namely FraSCAti) responsible for creating runtime components from SCA assembly definitions. The assembly factory is the front end component of the FraSCAti platform infrastructure. Its main component, the manager, drives the instantiation process by invoking parser, completion, validation and SCA component instantiation sub tasks. Part of the factory implementation is based on the Eclipse STP SCA meta model resulting from our work detailed in Section 3. The assembly factory relies on the Tinfi runtime to instantiate SCA components (see Section 4) and the binding factory to create remote connections (see Section 5).
In this part, we specify, the architecture and the implementation steps of the SCA assembly factory. First, we give details about the architecture of the factory and its internal components. Second, we specify the instantiation process performed by the runtime factory. Finally, we evaluate the SCA model implementation which allows the assembly factory to manipulate SCA assembly descriptions.

8.1.2 Architecture

In this section, we give an overview of the SCA assembly factory. As already mentioned, the assembly factory is responsible for reading assembly definitions and creating component instances using the runtime support provided by Tinfi. The runtime factory can act as an SCA domain manager, allowing to load several SCA composites.

The assembly factory is composed of five functional units :

- The assembly factory
- The model parser
- The completion component
- The validation engine
- The instantiation component

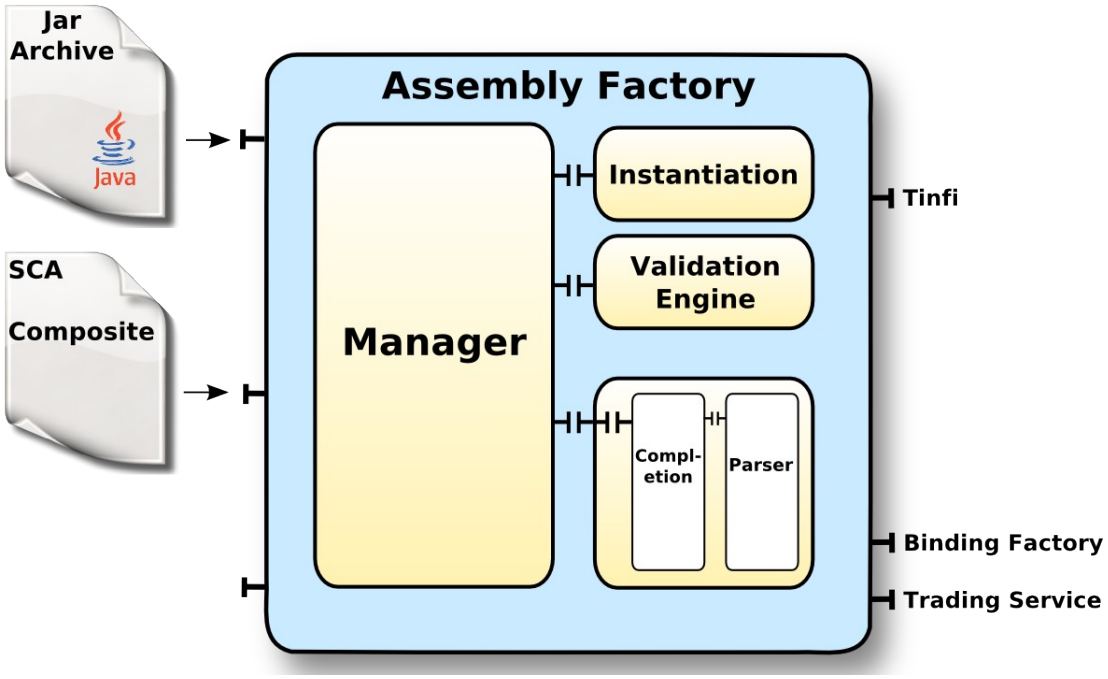


Figure 40: Assembly Factory architecture overview

a) Manager

The manager is the main component of the assembly factory. It is responsible for the management of the parser, completion, validation and instantiation process. The assembly factory can be configured in order to adapt the component execution flow. For instance, the assembly factory can ignore the validation process to gain performances (despite of safe loading), or just test composite file validity without performing instantiation. However, the basic assembly factory schedule remains a sequential execution of the parser, completion, validation and instantiation components.

b) Parser

The parser component is responsible for reading assembly definitions of an archive. It provides a reader for composite file and an implementation of the SCA assembly model. This implementation allows the reader to create a memory model from a composite. For the parser implementation we choose to rely on existing XML parsing tools. We evaluate XML technology for processing assembly definition in the implementation section (see 8.1.4), as we thought this choice has a major impact on assembly factory performances. The model implementation is based on our corrected XML schema of the SCA assembly model (see Section 3).
We identify four possible XML technologies to process SCA assembly descriptions:

- Streaming API for XML (StAX) combined with an SCA model implementation.
- Document Object Model (DOM) parser.
- Java XML Binding (JAXB).
- Modeling using the Eclipse Modeling Framework (EMF).

In Section 8.1.4, we discuss about constraints and benefits of each of these solutions.

c) Completion

The completion process gets annotations from a component implementation and completes the memory model generated by the parser. The SCA specification is composed of different documents. They define how to describe and implement an SCA application. When designing an application, some informations of the SCA specification are both defined in the description and the implementation. For instance, in the Java implementation specification, it is possible to define default property values while this information is not required in the assembly definition. In this case, the

default property values need to be extracted from the implementation, and added to the assembly description.
The completion is responsible for introspecting the SCA component implementation and extract annotations. When informations are missing in the memory model, it adds them to the corresponding component. Since the instantiation is based on the assembly description, this process is always required. Otherwise, some details relevant to the application can be missing when components are created and launched.

d) Validation

The validation engine checks structural properties of loaded data from SCA assemblies definitions and validate classes according to their implementation. The structural validation verifies assembly definitions according to rules given by the SCA Assembly model specification. By example verify that a component exists, for each component reference. These rules are listed in the part related to the meta model (see Section 3.3.3). Then, the completion component extracts annotations from implementations by using introspection and verify them with the loaded model.

e) Instantiation

The instantiation is the last stage of the assembly factory process. The instantiation objective is to visit the assembly description and create component instances. It uses the functionalities provided by the Tinfi runtime and the binding factory. The instantiation component uses the parser component to traverse the memory model. For each composite, the instantiation creates component types, service and reference interfaces and sub components. The instantiation main function is to traduce assembly description into method calls to the Tinfi runtime and the binding factory. As Tinfi implementation is based on the Fractal component framework, SCA components instantiation in FraSCAti is very close to Fractal components build method.

8.1.3 Specification

a) Composite file parser

The SCA Assembly specification describes the SCA assemblies model using the Extensible Markup Language (XML). It also provides a XML Schema which aims at define the SCA document format allowing to express SCA component assemblies.

The Extensible Markup Language (XML) benefits of many tools to read, write, handle documents. Using XML schema definition permits to define a class for XML documents which describes SCA assemblies. In this part, we compare solutions for loading SCA assemblies in memory with the SCA Assembly Factory.

Streaming API for XML

The Streaming API for XML (StAX) is an application programming interface which enables reading and writing XML documents. StAX is an intermediary solution between traditional event based XML parsers (Simple API for XML) and tree based XML parsers (Document Object Model, for instance). While SAX pushes events to applications when XML data is read, StaX allows the application to pull data as its convenience. The application keeps control over the parsing process by iterating over XML documents according to its needs. At any time, the StAX parser holds only a small part of the document, which makes StAX highly efficient for processing large documents.
StAX has been retained by the Apache Tuscany SCA project as solution for reading SCA assembly definitions. Actually, StAX provides the XML parser while Apache Tuscany SCA implements objects which represent SCA assembly into system memory. Therefore, a considerable effort is given to maintain code for manipulating SCA assembly. Changes to the SCA model specification need to be reported to the implementation. Verification of loaded SCA assemblies is also dependent from implementation. Without a model driven approach, code maintenance can be a repetitive and error-prone task.

Document Object Model

Document Object Model (DOM) [25] is a recommendation from the W3C. It provides a platform and a neutral language interface which enables programs to dynamically access and update documents. DOM enables to create tree-based data structures from (usually) XML documents. The data structure is stored into system memory, allowing programs to access and explore nodes to get associated informations.
In the SCOrWare project case, DOM provides an easy way to read and load SCA assemblies into system memory. Since DOM provides API to access any element in a XML document, SCA assembly descriptions can be easily read and written in a generic way. This allows to read any SCA composite files, even if some extension points (binding or

interface for instance) are not defined by a schema. However, lack of typing for SCA elements is a big drawback. As any SCA elements are threated as XML elements, each element must be tested to retrieve the SCA concept it refers to. This implies implementing a lot of control code which retrieves, the SCA concept associated with an XML node, and its parameters, before performing required task like validation or instantiation. To sum up, DOM provides a simply way to parse and load an XML document into memory.

Java XML Binding

Java XML Binding typically consists in associating a XML schema definition with Java classes. The Java Architecture

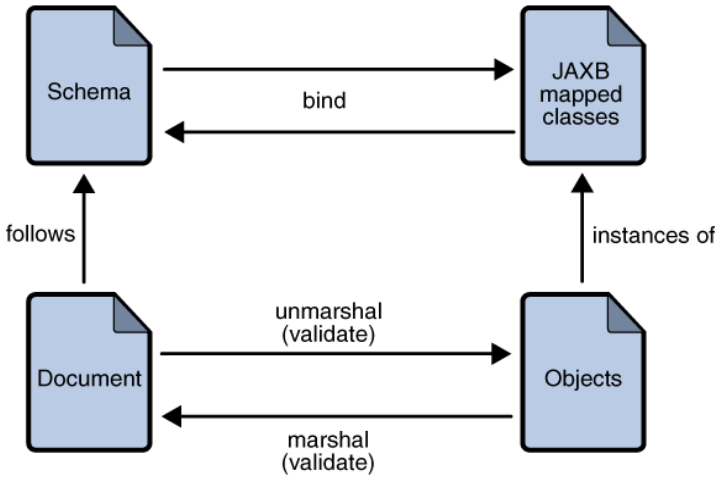


Figure 41: Marshaling / Unmarshaling process in JAXB

for XML Binding (JAXB) provides a binding compiler which enables generation of Java classes from XML schema definition. Therefore, the binding framework (generated and utility classes) allows marshaling and unmarshaling of XML document instances in a content Java tree. Additionally, XML instances can be validated while performing marshaling/unmarshaling operations (Fig. 41).

In the context of SCOrWare, the Java Architecture for XML Binding allows, with corrected SCA schema, to generate Java classes to parse, handle and load SCA assemblies. With the help of validation tool, we can check validity of loaded SCA assembly structure. If parsed assemblies are wrong, the validation tool reports the line and reason for each error. JAXB eases the loading process in memory, ensures validation of parsed assembly, and eases handling of SCA assembly. Moreover a change in the SCA model can be rapidly reported by applying modification to the schemas and generating corresponding Java classes. In comparison with StAX, JAXB doesn't need to implement StAX parser which needs specific (error-prone) processing for XML events. In comparison with DOM, JAXB is more efficient and avoid accessing data through a strongly typed tree structure.
Similarly to JAXB, Service Data Object (SDO) [11] provides a binding framework. While JAXB focuses on binding Java to XML, SDO goes one step further by enabling uniform access to heterogeneous data sources through its Data Mediator Service. SDO offers also static and dynamic API whereas JAXB only provides a static binding. As we focus on efficient SCA assembly processing written in XML, JAXB is, in our case, sufficient.

Modeling framework

Modeling framework aims at ease development of model-driven applications. Modeling allows developers to focus on application design instead of dealing with implementation and system details. They improve software reuse and maintainability.
The Eclipse Modeling Framework (EMF) provides tools for creating models, generates efficient and customizable Java code for creating, importing, reading, editing a model. EMF is based on a specific meta-model named Ecore which allows to describe EMF models. Ecore meta-model is defined by 17 meta classes (Fig. 42) which basically defines classes, references, attributes, data types, *etc*. The Ecore based models are serializable into Ecore XMI or Essential MOF XMI. The Eclipse Modeling Framework is also able to create Ecore models using XML schema definition. In the context of SCOrWare, EMF is a powerful tool which allows us to focus on the SCA model. It generates Java code to load SCA assembly in system memory as well as it generates graphical tool for editing and saving SCA assembly descriptions. The EMF generator supports merging regeneration which preserves manual change in the implementation.

Comparing to JAXB, EMF promotes a complete modeling solution which hides code complexity and generates tools for loading and handling SCA assembly.

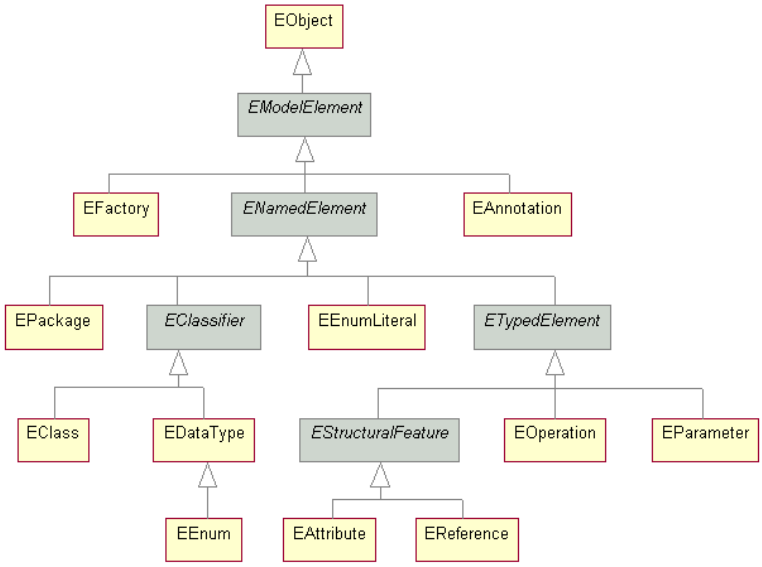


Figure 42: Ecore Model meta classes

Parser comparison

In order to choose a solution among techniques for parsing and loading SCA assembly, we need to analyze performance for each one, but also balance performance with code complexity and tool capabilities. In SCOrWare platform case, we focus on an assembly factory which verifies and ensures correctness of SCA assemblies. Performance tests evaluate memory and CPU consumption when creating data structure instances, parsing and validating assembly definition. To do that, we have established a small testbed we have applied on each solution. This testbed is composed of a sets of assembly descriptions which are given as input sources for memory loading. Used assemblies are extracted from samples available in the Apache Tuscany SCA 0.90 incubating distribution. Our performance evaluation focuses on code weight, memory and CPU consumption.

We evaluate overall performance of each solution using Java unit tests. These tests load our set of SCA assembly descriptions with StAX, DOM, JAXB and EMF implementations and report average execution time in the next diagrams. The first diagram (Fig. 43) compares time taken to load five assembly descriptions without validation. We can observe that the StAX implementation used by the Apache Tuscany SCA project is faster than JAXB and EMF solutions. Comparing to DOM, Apache Tuscany SCA load times seem to be equivalent. Apache Tuscany loads SCA assembly descriptions into corresponding Java objects, while DOM only enables handling of XML elements. In the second diagrams (Fig. 44) we activate validation of SCA assemblies. We discover that loading and validating SCA assemblies with DOM and JAXB dramatically increases execution time, while EMF is not much affected. This difference is particularly due to the XML schema definition. Actually, DOM and JAXB rely on XML schema definition to perform SCA assembly validation, which can be memory and time consuming, whereas EMF relies on the Ecore model allowing to generate Java code to validate SCA assemblies, and avoid additional XML processing. Finally, in spite of our efforts to validate SCA assemblies locally (XSD and DTD referenced locally), we observe that validation with DOM and JAXB always requires a network connection. This limitation can be an issue for SCA applications in local networks. Actually, it is possible to build catalogs of XML schema, but as we only want to evaluate each solution in the context of the SCOrWare platform, we haven't tried to perform further tests. As the Apache Tuscany SCA implementation doesn't validate SCA assemblies we only compare DOM, JAXB and EMF. Validation within Tuscany SCA is a work in progress, so we will be able to compare it with DOM, JAXB and EMF. According to StAX validation which relies on XML schema, we think that validation within Tuscany SCA will be comparable as DOM and JAXB.

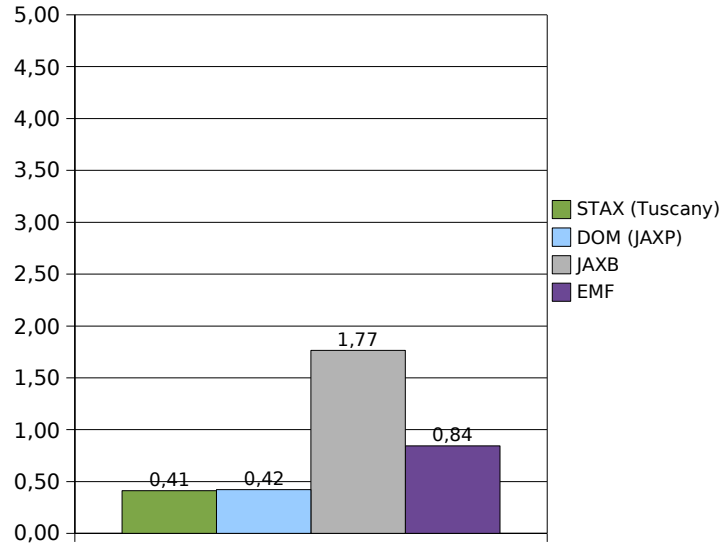


Figure 43: SCA loading without validation

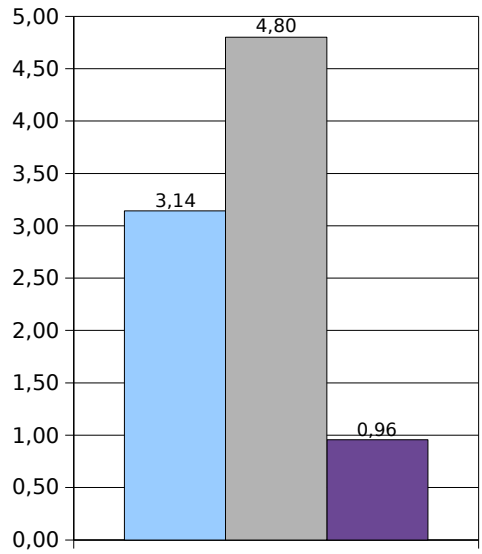


Figure 44:SCA loading with validation

In a second part we compare package size and memory footprint for DOM, JAXB or EMF. Figure 45 reports memory usage variation given by the Java virtual machine. With this diagram, we can observe that comparing to EMF and DOM, the Java architecture for XML binding consumes much more system memory. We denote also in Figure 46 the size of generated code which allows to manipulate SCA assemblies. While the Eclipse Modeling Framework generated code size is more important, it seems to provide a memory and CPU efficient solution to process and validate SCA assemblies.

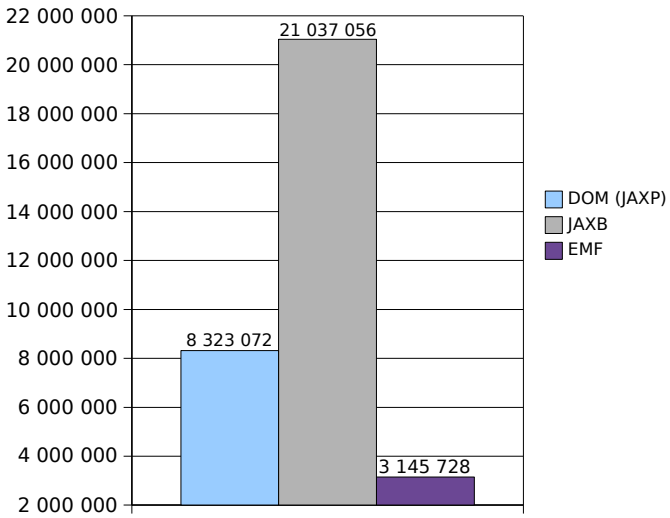


Figure 45: System memory variation

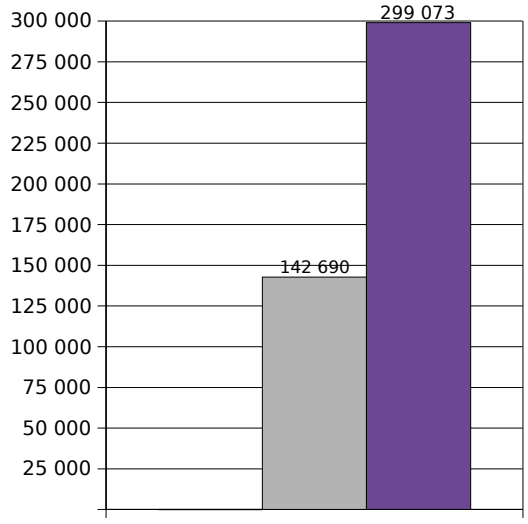


Figure 46: Package Size

Looking into generated code details, we perceived that EMF generated code is better structured. It dissociates interface and implementation for objects describing SCA assembly and provides a set of utility classes to marshal, unmarshal, validate documents. Contrary to JAXB which maps XML elements to dedicated objects, each SCA object generated with EMF is defined as an extension of the Ecore model. This mapping between Ecore and generated code allow us to easily traverse SCA assembly using generic objects (like DOM) or dedicated (like JAXB).
In this section, we have described different solutions which enable the SCOrWare platform to load and process SCA assembly definitions. Although the Apache Tuscany SCA platform provides an efficient method for processing SCA assembly, it doesn't verify structure and constraint of the SCA assembly model. Apache Tuscany relies SCA on StAX, which is the most efficient solution for parsing XML files. But StAX proposes nothing more than parsing. Moreover the implementation needs to be maintained by developers to take into account changes in the SCA model. As a result, Apache Tuscany SCA implementation allows to load assembly descriptions which can contain errors (missing attributes as example), and then introduce exception during component instantiation and runtime. In opposite, we have tried to manipulate SCA assembly using different XML tools : DOM, JAXB, and EMF. While we have demonstrated that these tools introduce an additional cost, we have also evaluated cost added when we use SCA assembly validation (not supported by Apache Tuscany SCA).
The SCOrWare project aims at providing a platform which validates SCA assemblies to ensure that loaded component architectures are correct. Considering our evaluation, the Eclipse Modeling Framework provides an efficient model driven approach to load, edit, write, verify SCA assemblies. EMF allows us to focus on the SCA model while it provides tools to generate code which permits to instantiate an assembly into system memory. Comparing to the StAX solution used in Apache Tuscany SCA, our tests have shown that using EMF introduces a small cost while it benefits from the model driven approach. This allows us to easily manipulate SCA assembly from low-level implementation to high-level design tools.

b) SCA Component Instantiation

The assembly factory main objective is to read assembly descriptions in order to build composites and their components. After having parsed and verified an assembly description, the assembly factory uses the Tinfi membrane control interfaces to create components. As Tinfi is implemented *via* the Fractal component framework [7], using Tinfi for creating components, typically consists in using interfaces described in the Fractal API. Thus, create SCA components with Tinfi is similar to create Fractal components. By analogy to SCA, Fractal ADL provides a XML based language for describing Fractal assemblies, and a factory to create components. As a result the Fractal ADL factory represents a starting point for the SCA assembly factory in SCOrWare. This part mainly describes how the assembly factory process composites files and how it can invoke Tinfi and the binding factory to create component instances and link their interfaces.

Create SCA Composites

The assembly factory loads SCA composite descriptions using the generated EMF model from the corrected schema of the SCA specification. In order to create a composite and its components, the assembly factory traverses the loaded SCA composite in memory, and initializes, configures, binds and starts components. Traversing loaded composites is done by the instantiation component. When the instantiation component reads an element in the assembly description (a component, for instance), it performs a list of corresponding actions. For a component element, the instantiation will first create a service type, a reference type and a component type. Then, it will create the component instance with the corresponding service and reference interfaces. Finally it will set the component name and its properties.
First of all, creating components using Fractal API requires to bootstrap Tinfi runtime and gets its factories. The bootstrap is obtain from the `Fractal.getBootstrapComponent()` method. It provides the Tinfi bootstrap component which enables to create other components. The returned component provides two interfaces : the generic factory interface and the type factory interface. The former allows to create new component instances returning the component interface while the later permits to create component and interface types. The initialization code is written below.

```
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.factory.GenericFactory;
import org.objectweb.fractal.api.type.ComponentType;
import org.objectweb.fractal.api.type.InterfaceType;
import org.objectweb.fractal.api.type.TypeFactory;
Component tinfi = Fractal.getBootstrapComponent();
GenericFactory gf  = Fractal.getGenericFactory(tinfi);
TypeFactory tf  = Fractal.getTypeFactory(tinfi);
```

With this minimal initialization code, we are able to create SCA composites and components. Using the generic factory, the type factory, the control interfaces provided by Tinfi, and the model loaded by EMF, the instantiation component can traverse the SCA composite model. It invokes corresponding methods described by the Fractal API to create runtime assembly. In this part, we describe procedures which must be performed by the instantiation component to create SCA composites and its components.

- **Dealing with multiplicities**

The SCA model allows to define the number of wires that connects a reference to a target service. It indicates if the wire is optional or not and if the reference can have multiple wires. Similarly, in the Fractal specification, it is possible to define if the functionality provided by an interface is guaranteed and if the component has one or many interfaces of this type. The SCA specification defines four multiplicities (`0..1`, `1..1`, `0..n`, `1..n`) which corresponds both to the contingency and the cardinality of the Fractal interface type. When an SCA component or composite which has a reference with multiplicity attribute is created, the instantiation component has to convert the reference multiplicity into the corresponding contingency and cardinality of the Fractal component interface. The following table gives equivalences between SCA multiplicities and contingency cardinality in the Fractal type system.

SCA	Fractal	
Multiplicity	Contingency	Cardinality
0..1	optional	singleton
1..1	mandatory	singleton
0..n	optional	collection
1..n	mandatory	collection

Tableau 6: SCA multiplicities and Fractal corresponding types

- **Create a composite**

The SCA assembly model permits to describe one composite by each assembly description file. Therefore, the composite element is the starting point for any assembly description, and then, for the instantiation component. The

instantiation follows the composite description and performs component creation task according to a specific semantic described in the following lines.
Primarily, the instantiation component must determine the composite type. This means for each service and for each reference exported by the composite, the instantiation constructs the corresponding Fractal interface type using the Fractal type factory. For instance with a service described as below

```
<service name="s">
  <interface.java class="I"/>
</service>
```

corresponds to

```
InterfaceType serviceType = tf.createFcItfType("s", "I", server, mandatory,
singleton);
```

With
`s` - the name of the service
`I` - Java interface for this service
`server` - A provided interface
`mandatory`, `singleton` – settings for multiplicity 1..1

Sometimes, the interface is not specified for a composite reference or a composite service, because it is implicitly the interface of the promoted component. In this case the visitor has to refer to the interface of the promoted component (which is also optional). When interface of the promoted component is not specified in the assembly description, we have to introspect the implementation using the Java reflexive API to get its Java interface. This control is ensured by the completion process, performed before validation of the assembly and the instantiation process. Sometimes the instantiation component has to create an interface from a composite reference with a multiplicity attribute, it converts the multiplicity into the contingency and cardinality of the Fractal interface. For instance, with a reference with a multiplicity "0..n" :

```
<reference name="r" multiplicity="0..n">
  <interface.java class="I"/>
</reference>
```

corresponds to

```
InterfaceType referenceType = tf.createFcItfType("r", "I", client, optional,
collection);
```

With
`r` - the name of the reference
`I` - Java interface for this reference
`client` - A provided interface
`optional`, `collection` - settings for multiplicity 0..n

Multiplicity must be equivalent to the contingency and cardinality parameters given when calling the `createFcItfType()` method with the type factory. Correspondences between multiplicities and contingency cardinality is given in the previous section.

When service and reference interface types constructed, the instantiation component can now determine the composite type. For instance when having a composite with one or more services / references :

```
<composite>
  <service .../>*
  <reference .../>*
</composite>
```

the instantiation executes

```
ComponentType compositeType = tf.createFcType(
                                new InterfaceType[] {
                                  s1Type, ... snType, r1Type,... rnType
                                }
                              );
```

With
`s1Type, ... snType` – list of service interface types
`r1Type, ... rnType` – list of reference interface types

This second procedure creates, a new component type according to the interface types which has just been determined for each service and reference. The procedure `createFcType()` provided by the type factory returns a new component type. This component type, according to the Fractal type system, describes a set of component interface types. A component type in Fractal is represented *via* the Java `ComponentType` interface and a component interface is represented *via* the `InterfaceType` interface. The `ComponentType` interface defines operations which return the set of component interface types or get a particular component interface, while the `InterfaceType` describes an interface with its name, its signature, its role, contingency and cardinality.

We can now build a new composite instance with the created component type. Creating a new composite instance is realized with the generic factory. This is characterized by the code below :

```
Component composite = gf.newFcInstance(compositeType, "scaComposite", null);
```

Currently, we have only produced an empty composite instance and its interfaces. Now, the instantiation component has to inspect the content of the composite description and get component definition inside. Once components of the composite are defined (described in the next section), the instantiation adds the components to the containing composite. To do this, we get the content controller interface provided by our composite with the Fractal API. The corresponding description :

```
<composite>
  <component ...>
</composite>
```

implies

```
Fractal.getContentController(composite).addFcSubComponent(component);
```

The instantiation component iterates over the collection of components and adds each component as a sub component of a composite. For each component, we have to call the `addFcSubComponent()` method provided by the content controller of the Fractal API. When the composite and its components are created, we have to set links in respect with the architecture description of the assembly. In the SCA specification, we can distinguish tree types of links : (i) links between component service and component reference, (ii) links between composite and promoted components, (iii) wire between components. Create components binding is defined in next paragraph.

- **Create a component**

According to the Service Component Architecture, a composite contains one or more components. Thus, creating composite assembly needs also to create, link, and configure components. We describe in this part, how the instantiation process creates components using Tinfi.
First, the instantiation has to determine the interface and component types. Likewise a composite, a component can describes its services and its references. The former defines functionalities provided by the component while the later defines dependencies to other components of the composite. As for composite interfaces, the instantiation component uses the type factory to creates interfaces of the component. For instance, when a component describes its services :

```
<service name="s" multiplicity="0..n">
  <interface.java class="I"/>
</service>
```

Will result in execution of the following code

```
InterfaceType serviceType = tf.createFcItfType("s", "I", server, optional,
collection);
```

We can notice that building interface types of an SCA component is same as building interface types of an SCA composite. Actually, this is due to the Fractal component model which enables to describe components as assembly of components. In other words, this demonstrates in practice that both the concepts of component and composite from the SCA model are equivalent to the concept of component of the Fractal model. Then the instantiation component creates the component type. Like processing interface types, this operation does not differ with creation of a composite type. Thus we can refer to the example showing how to build the composite type as example for building a component type of an SCA component.

Once component type has been determined, we can create a component instance. This is done using the generic factory, through the following code :

```
Component component = gf.newFcInstance(componentType, "scaPrimitive", class)
```

Contrary to a composite, when creating a component we have to refer to the code which implements the component functionalities (represented by the `class` parameter in the previous example). In the SCA assembly model, the component implementation can be written using many languages and technologies like Java, BPEL, C++, *etc*. As we currently focus on Java implementation, we just have to distinguish when component is directly implemented in Java (`implementation.java`) or when its implementation is a composite (`implementation.composite`).

If the component description defines that the component is implemented with a Java class (named `C` for instance)

```
<component>
  <implementation.java class="C"/>
</component>
```

then the instantiation component extracts the class attribute from the description and performs the following Java code

```
Component component = gf.newFcInstance(componentType, "scaPrimitive", "C");
```

But if the component implementation is defined as a composite like :

```
<component>
  <implementation.composite name="namespace:compositeName"/>
</component>
```

The instantiation process must call the assembly factory to process the assembly description of the referenced composite name. When the referenced composite file has been processed, the assembly factory will return the instantiated composite to the original assembly factory. Then, the returned SCA composite, corresponding to an composite implementation of a component, can be added as a subcomponent of the current composite.

Finally we have to set a name for the built component, and configure it. The Fractal Name Controller allows us to associate names to components. For example, if we have the following component described in the assembly :

```
<component name="N">
  ...
</component>
```

The instantiation process sets the component name with :

```
Fractal.getNameController(component).setFcName("N");
```

Then, the component properties must be configured. The configuration of an SCA component through its properties is done with a property controller specific to Tinfi. In SCA, a property can be defined in the component or in the composite description. When a value for a property is defined in a composite, a reference to this value is described by a component. For instance if the value is given as a component property :

```
<component name="N">
  <property name="property">
  value
  </property>
</component>
```

The instantiation process sets the component attribute with :

```
component.getFcInterface("sca-property-controller").set(property,value);
```

The instantiation component gets the property controller interface of the component and sets the property according to the given name and value in the assembly definition. In order to set the property value, the instantiation component has to convert the XML schema type of the value into a Java type. If the value of the component is given in a composite element, then the visitor has to retrieve the value of the property indicated by the "source" attribute.

```
<composite name="C">
 <property name="sourceProperty">
  value
 </property>
<component name="N">
  <property name="property" source="$sourceProperty">
</component>
```

The instantiation process sets the component attribute with :

```
component.getFcInterface("sca-property-controller").set(property,value);
```

- **Create local binding**

In the previous paragraph we have described how to create SCA components and composites, using the generic factory and the type factory provided by Tinfi. While we give details for creating component instances, we don't know how to assemble components. In the SCA assembly model specification, components and composites are composed using bindings defined by component (or composite) reference and service. The SCA assembly model allows to bind components using multiple communication protocols. Binding between local components, in the FraSCAti platform, is done by using Fractal API while remote bindings are ensured by the binding factory (described in Section 5). According to the SCA model, we distinct three ways to bind SCA components : (i) by defining target service of a component reference, (ii) by defining a wire between a component reference and a component service, (iii) by promoting a component reference or a component service (binding between composite and component interfaces).

In the first case, when visitor reads component definition and gets a target attribute from the component reference, it has to create a binding from the component reference to the required service. In the SCA terminology, this is defined by the following code example :

```
<composite>
 <component name="c1">
   <reference name="r" target="c2/s"/>
 </component>
 <component name="c2">
   <service name="s"/>
```

```
 </component>
</composite>
```

The following code will be executed in order to create link between them

```
Fractal.getBindingController(c1).bindFc("r",c2.getFcInterface("s"));
```

In a second case, the wire definition is, in the SCA model, another way to express links between components. Thus the wire definition is similar to the previous case. The next example shows the code executed by the visitor resulting from description of a wire. When the assembly defines :

```
<wire source="sourceComponent/referenceName"
target="targetComponent/serviceName"/>
```

The visitor component will execute

```
Fractal.getBindingController(sourceComponent)
 .bindFc("referenceName",targetComponent.getFcInterface("serviceName"));
```

The third case, when the assembly defines composite which promotes a service or reference, the visitor binds the composite with the component service/reference interface and exports the service or binds the reference with `hints` informations defined by the communication protocol parameters.

```
<composite>
  <service name="s1" promote="c/s2">
  <component name="c">
    <service name="s2">
  </component>
</composite>
```

Then the visitor will create a binding between the service offered by the composite and the service offered by the promoted component. This is traduced by the following code

```
Fractal.getBindingController(composite)
  .bindFc("s1",c.getFcInterface("s2"));
```

- **Create remote binding**

The following code example shows how to use the binding factory. Typically remote binding between client and service interface is done in two steps. Firstly, we export the service interface with the `export()` method offered by the binding factory. Then we bind the client with the service interfaces. When the binding factory is used to export or bind interfaces, the `export()` and `bind()` methods are called using a `hints` argument which list specific parameters for the used binding protocol. `Hints` parameters are string value pairs extracted from binding attributes given in the assembly definition.

```
<composite>
  ...
  <component name="c">
    <service name="s">
     <binding.[protocol] [hint]="value" />
    </service>
  </component>
</composite>
```

Then the visitor will export the service offered by the component. This is traduced by the following code

```
bf.export(component, "s", hints);
```

The next tables give possible hints pair according to binding protocols defined in the SCA specification.

Export Mode value	"rmi"

Parameter name	Parameter Value
host	Host name where the RMI registry is located
port	Port of RMI registry
rmiservice	Name of the service in the registry

Table 7: Hints parameters for RMI <binding.rmi>

Export Mode value	"ws"

Parameter name	Parameter Value
endpoint	Endpoint of a service or a reference
element	Specify URI of a WSDL element
location	Location of the WSDL document

Tableau 8: Hints parameters for SOAP <binding.ws>

8.1.4 Implementation

This part gives implementation details of the assembly factory. While the assembly factory is responsible for building SCA component instances, it is itself implemented using the Fractal component model. Thank to the Fractal component model, the assembly factory is entirely configurable. For instance, this allows to easily extend the instantiation process by adding support for new binding type, or activating / deactivating model validation according to application needs (performance vs correctness).
The assembly factory implementation can be decomposed in three different personalities : the core factory, the generate factory and the runtime factory.
First, the core factory is the main module which provides the ability to create SCA component instances. SCA components are generated at compile time. As a consequence, the core factory can only create component instances using SCA components previously generated and compiled with the Tinfi compiler. Second, the generate factory extends the core factory to allow to generate and compile SCA component implementation by using the Tinfi compiler. It does not perform component instantiation. Finally the runtime factory combine both the functionalities offered by the core and the generate factory to allow user to generate and immediately instantiate SCA components from a Java archive. Although most of the SCA users will prefer simplicity and use the FraSCAti runtime platform; Advanced FraSCAti users should appreciate the possibility to avoid component code generation at runtime (using the generate factory to compile the SCA application) which consumes more system resources.
In this part, we will briefly discuss on differences between assembly factory personalities. Then we will detail the runtime assembly factory, more precisely the internal components responsible for managing the assembly factory and creating SCA composite instances. Finally we deal with the possible extension of the assembly factory in order to allow processing of SCA composite file with new implementation or binding type.

a) Core, Generate and Runtime personalities

As already mention, the assembly factory is decoupled in three distinct personalities : the core factory, the generate factory and the runtime factory. The core assembly factory personality is the main module which allows to create component instance while the generate factory is responsible for generating SCA Java components implementation. Additionally the runtime factory combines generation and instantiation in order to ease assembly factory usage.

The main difference between those personalities is the implementation of the component responsible for invoking the kernel. When the assembly factory creates component instance (core factory), the runtime component invokes Tinfi runtime kernel; When the assembly factory generates component implementation (generate factory), the runtime component invokes the Tinfi compiler. At runtime, both Tinfi compiler and runtime kernel are invoked.
The Tinfi component compiler needs to be initialized with some parameters, For instance, source and target directories. Thus, the assembly factory for component generation provides an initialization interface in addition to the component build interface given by the core factory.

In practice, the three personalities of the assembly factory are described by three different (but similar) Fractal component definitions. The assembly factory provides a single entry point which allows to get an instance of a particular personality (choosing the appropriate architecture to instantiate).
In the following part, we give details on internal process and components of the assembly factory, given in the figure 40. We will focus on the runtime personality.

b) Manager Component

The manager component is a front-end for the assembly factory. It provides the assembly factory main entry point which allows to load SCA applications from SCA composite definition. The manager is also responsible for driving assembly factory internal components.
When the assembly factory is requested for loading an SCA composite (see figure 47), the manager component calls the composite file parser to create an SCA model instance (using the composite definition). Since part of the SCA component definition can be defined in components implementation, the resulting model may be incomplete. For instance, SCA components implemented in Java can use annotations to describe service interfaces, references, properties, etc. In such case, the parser component calls the component responsible for completing the model with data defined by the implementation. This completion component currently supports introspection of SCA components implemented in Java (using the reflection API).

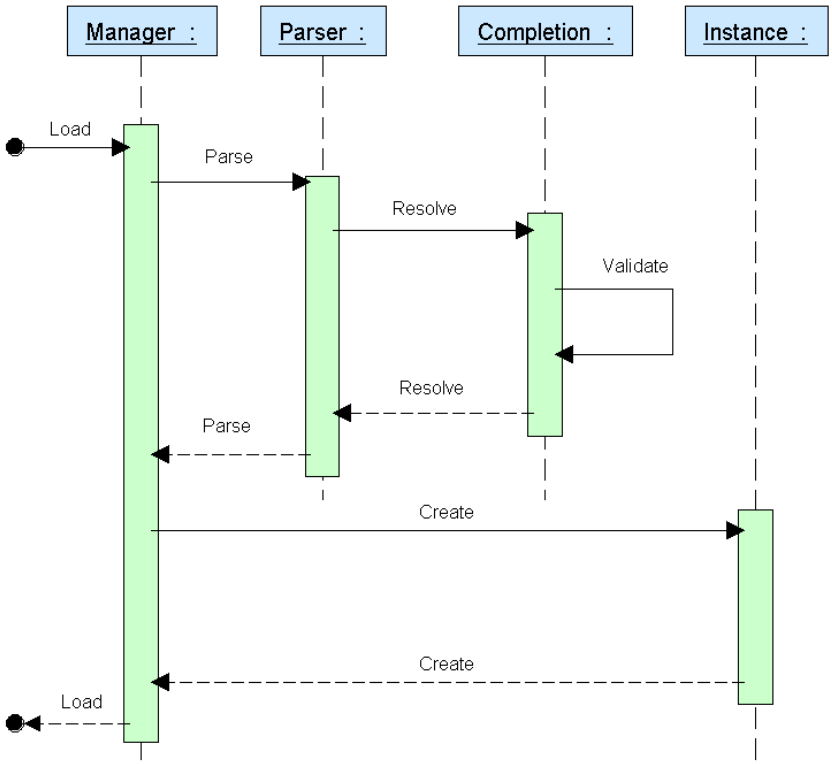


Figure 47: SCA application loading sequence

Once the composite description has been parsed and the composite model instance completed, the manager can use the model instance to instantiate the SCA composite. The manager invokes the instantiation component responsible for interpreting the SCA model and using the Tinfi kernel for creating SCA components instances. We give details of the assembly factory sub components in the next sections.

c) *Parser component*

The parser component of the assembly factory is mainly responsible for loading the composite definition. This component is based on the SCA model provided by the STP SCA project from the Eclipse development platform. The parser component first uses the SCA xml parser provided by STP to create an instance of the composite definition. The resulting model instance allows us to navigate easily through composite definition, however as already discussed, part of the model description can be directly defined in components implementation. Thus, the parser component has a sub component dedicated for resolving data from component implementation. Actually the resolver component supports introspection for SCA components implemented with Java. The resolve method helps in retrieving the Java interfaces for component services and component references and link components according to the target and promote attributes. Once the composite model has been resolved it is validated using the method provided by the EMF framework, this ensures that the composite model given to the instantiation process conform to the SCA specification.

d) *Instantiation Component*

The instantiation component is responsible for interpreting SCA composite model instances into invocations of the Tinfi kernel and the binding factory. The instantiation component is composed by seven sub components responsible for processing different parts of the SCA composite model (see figure 48). The Assembly and Component components respectively process SCA composites and SCA components according to the SCA composite model instance. For each component defined in the model instance, the component builder set components interfaces, sets the component implementation, intents, properties and creates bindings on SCA services and references.
The instantiation component is implemented according to specification of the “SCA Component Instantiation” described in section 8.1.3

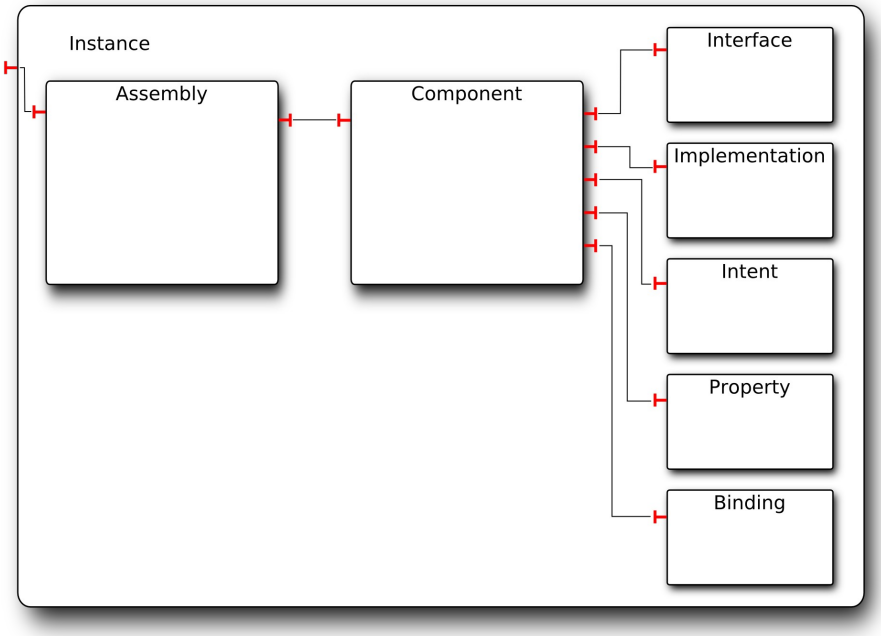


Figure 48: Instantiation Component

The figure 49 shows the sequence diagram for the instantiation process. Firstly, when the assembly factory instantiation component is requested for building a new SCA composite component, the assembly component creates a container. This container basically helps in managing SCA composites individually. For example, the container provides method for starting or stopping SCA composites. Next, the assembly builder delegates the instantiation of the composite component to the component builder. The SCA component builder creates the SCA composite instance, sets the name of this composite, then, for each SCA component defined in the composite model, invokes the component build method. The component build method uses the interface, implementation, intent, property and binding components, to create an SCA component instance. The component instantiation method creates the component type from component interfaces defined in the model. Then, using the component type, the implementation component creates the SCA component from the specified implementation. For instance, in case of Java implementation, the component will invoke the Tinfi kernel. Once the component instance has been created, the component builder sets additional data like intents and component properties. Currently, the assembly factory supports transaction intents. The component responsible for setting such intents on SCA components uses the transaction manager provided by the FraSCAti platform (see Transaction Service, Chapter 6). Finally, when SCA components defined in the current model have been created, the composite build method constructs bindings for composite/component services and references. Wires between components of the current composite are created first; Then promoted services and references are processed. Bindings using remote communication protocols are generated and initialized by the Fractal Binding Factory at the end of the composite instantiation process.

Note that intents, properties or bindings are not processed when SCA component are generated, these data are only relevant when the SCA composite is instantiated.
At the end of the instantiation process, the SCA composite is returned to the manager component. Since the FraSCAti platform is based on the Fractal component model, the SCA composite assembly provides control interface for managing the component life cycle. Thus at the end, the manager uses the life cycle control interface to start the composite, allowing the composite to emit or receive invocations.

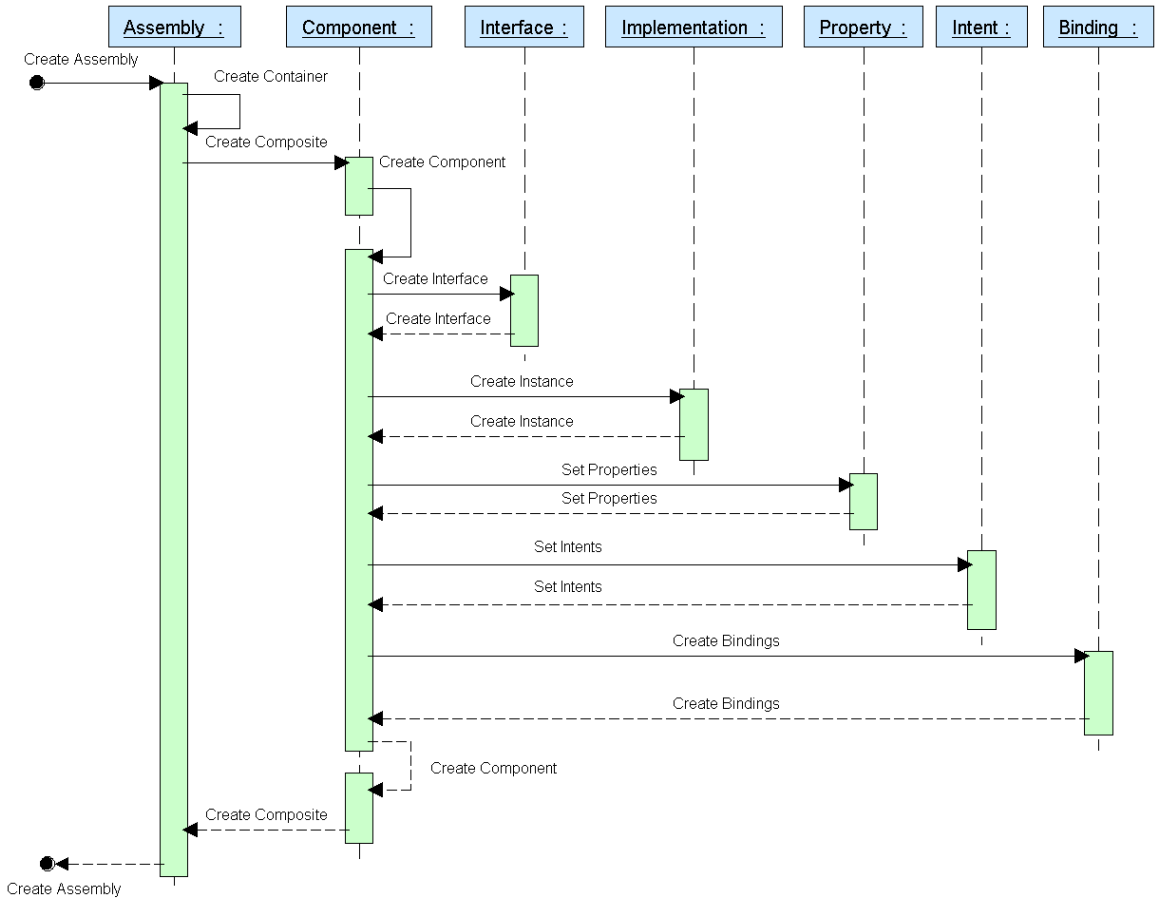


Figure 49: SCA composite instantiation sequence

e) *Extending the Assembly Factory*

The assembly factory has been implemented using the Fractal component model. Since a component architecture defined with the Fractal model can be extended, it is possible to extend the assembly factory. For instance this allows FraSCAti to support new implementation or binding type. Moreover, the implementation and binding components have been designed to be plug able. Concretely, the implementation and binding components are composed of sub components responsible for choosing the appropriate implementation/binding type; and a set of plugin components dedicated to a particular binding/implementation support (see Figure 50). The switch component checks the requested implementation/binding type with the available plugins. If a plugin for the implementation/binding type exists, the method call is delegated to this plugin, otherwise an error is raised and the instantiation process stops. To extend the assembly factory, developers have to :

- Implement a plugin component reflecting the plugin implementation / binding interface
- Add this component to the assembly factory architecture

Adding Binding Plugin

Actually, the assembly factory supports RMI and SOAP bindings between SCA components. In order to process new binding type (Java Messaging Service, for instance), plugin developers have to implement the binding plugin component which exposes the interface named “SCABindingProtocol”. This interface define three methods :

– `getBindingID()`, Allow to retrieve the binding identifier. This method returns an integer corresponding to the identifier of the binding type as given in the Eclipse STP SCA model. This identifier is used by the “Switch” component to choose the appropriate binding plugin.

- *bind(Reference, Binding, Component)*, Provide the ability to connect an SCA component reference using the specified binding on the given Tinfi component.

- *export(Service, Binding, Component)*, Provide the ability to export an SCA component service using the specified binding on the given Tinfi component.

Since the FraSCAti platform uses the Fractal binding factory, binding or exporting an SCA component typically consists in invoking the binding factory giving a list of suitable binding hints. Usually binding hints are extracted from binding attributes in the composite model. However adding support for a new binding protocol in the FraSCAti platform both requires to define this binding in the composite model, adds a plugin in the assembly factory and adds plugin (implementing the requested protocol) in the binding factory.

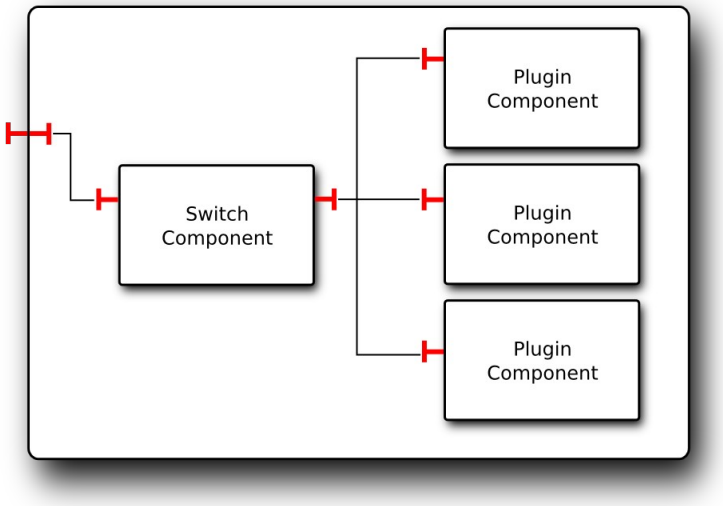


Figure 50: Plugin Architecture

Adding Implementation Plugin

As Binding, Implementation types are implemented as plugin in the architecture of the assembly factory (according figure 50). The current FraSCAti platform provides implementation for composite, Java, Spring and Fractal components. A plugin for a specific implementation has to provide the following methods :

- `getImplementationID()`, Give the identifier for the implementation plugin. This identifier corresponds to the implementation type defined in the STP SCA composite model.

- `create(Implementation, ComponentType)`, Returns an SCA component instance for the specified implementation type. Due to the Fractal component model, taken as basis for the implementation of SCA components, the interface assumes that returned object are of type “Component” of the Fractal API. As a consequence, each type of implementation has to be encapsulated into a Fractal component.

Add Plugin into architecture

While implementation / binding plugins can be packaged separately, adding plugins in the assembly factory requires to update part of its architecture. In the current revision of the assembly factory, the component architecture remains static. To add a binding plugin we have to update the Fractal definition of the binding component.
For instance if we want to add support for processing web service binding, we have to define a new Fractal component giving its name and its Java implementation, then we have to link the binding plugin with the component responsible for choosing the appropriate plugin and with the binding factory interface. In the following piece of code, we can see part of the actual Fractal definition of the binding component.

```
 1 <definition name="org.scorware.assembly.instance.binding.Binding">
...
 2   <component name="switcher"
 3     definition="org.scorware.assembly.instance.binding.ScaBindingSwitch" />
 4
 5   <component name="webservice"
 6     definition="org.scorware.assembly.instance.binding.ScaBindingWs" />
 7   <binding client="switcher.bindings-ws" server="webservice.binding-protocol" />
 8   <binding client="webservice.binding-factory" server="this.binding-factory" />
...
10 </definition>
```

The first Fractal binding (line 7) defines link between webservice plugin and the switcher component. The switcher component provides an optional client interface used to access to available plugins. The name for this interface is arbitrary, but, must begin by ***"bindings-"***. Inversely, the plugin component has to offer the Fractal server interface named ***"binding-protocol"***.

Adding an implementation plugin can be done by following this method for updating the implementation component of the assembly factory architecture. For instance the next Fractal ADL description is part of the definition of the implementation component with the implementation plugin allowing to support SCA components implemented with Java.

```
 1 <definition name="org.scorware.assembly.instance.implementation.Implementation">
...
 2   <component name="switcher"
 3   definition="org.scorware.assembly.instance.implementation.ScaImplementationSwitch"
 4   />
 5   <component name="java"
 6     definition="org.scorware.assembly.instance.implementation.ScaJavaImpl" />
 7
 8   <binding client="switcher.implementations-java" server="java.implementation" />
 9   <binding client="java.runtime" server="this.runtime" />
...
10  </definition>
```

In this definition the component named “java” (line 5) is responsible for processing SCA Java component in the composite definition. This plugin component is connected to the switcher responsible for choosing the appropriate implementation plugin (line 8). Note that implementation plugin always provides the interface named “implementation” which allows the switcher component to delegate invocations to the plugin. Additionally, the Java plugin uses the Tinfi kernel through the runtime interface (line 9).
The assembly factory architecture allows to support new implementation / binding type by adding plugin, but, it is not able to resolve plug-in dynamically. The static definition for the architecture of the assembly factory remain as an issue for dynamic loading of binding or implementation libraries.

f) Summary

This part was focused on the SCA assembly factory design and its implementation. The assembly factory is mainly responsible for interpreting assembly description and creating components. This factory is itself composed of different functional units which load, complete, verify and instantiate components. In this part the architecture gives an overview of the assembly factory, while the specification and the implementation part explain how the factory can instantiate components using the Fractal API. Since the Assembly factory is also implemented with Fractal, it is possible to reconfigure its architecture. For instance this allows to add support for new binding or implementation types. However, this actually requires to statically configure the assembly factory architecture.

8.2 System Deployment

8.2.1 Objectives

The Service Component Architecture describes a programming model which facilitates development of applications for distributed systems. The SCA model mixes concepts inherited from component- and service-oriented architectures. Business logic is implemented using tightly coupled and reusable components. When they are assembled, these components constitute a higher-level application which can be accessed through remote communications. The Service

Component Architecture is, in particular, able to support different (i) implementation languages and (ii) binding protocols on (iii) different application servers. While SCA focuses on describing software architecture, it doesn't care with the underlying physical infrastructure. According to the SCA philosophy, the targeted infrastructure is supposed to be known during application design. Thus, SCA applications are usually designed for a specific system infrastructure, the assembly descriptions (and particularly binding protocols) contain informations about service hosts and client hosts, or about communication ports which depend on the physical infrastructure. As a result, an SCA application depends on a specific system architecture, but doesn't give any information about it.
In the context of the SCOrWare project, we aims at providing a solution named Fractal Deployment Framework (FDF) [3] [8], which allows to specify how SCA applications are deployed on the underlying physical infrastructure. It allows to define hosts, SCA servers and SCA domains involved in an SCA system.

8.2.2 Specification

a) FDF overview

The Fractal Deployment Framework [3] is a generic component-based framework which aims at abstracting the deployment process of distributed systems. It allows to deploy several software layers, from the operating system to the top application level, while it enables parallel deployment on heterogeneous and large scaled distributed systems. FDF reifies as Fractal components every concept needed to perform a deployment, at a very fine grain. The assembly of these components abstracts the deployment process of a system on a physical architecture. The Fractal Deployment Framework is composed of three parts. Firstly, FDF provides a high level language which permits to describe deployment of systems. Secondly, it is composed of a set of software personalities which are executed by the deployment engine. Finally, FDF provides a library of primitive deployment components that encapsulate low level mechanisms.

FDF deployment language

FDF provides a high level language comparable to a kind of scripting language. The FDF language allows system administrator to describe easily the system architecture and software to deploy. Typically, a FDF deployment description defines the hosts of the physical architecture by declaring hosts' names, logins, file transfer and remote access protocols. Then, the FDF deployment file describes software to install using personalities. The administrator defines for a software the host on which it must be installed, the archive of this software, the installation directory. FDF deployment files are declarative.

Software personalities

The Fractal Deployment Framework is composed of many software personalities. These personalities defines how to deploy a specific software. A software personality is provided by a software expert who both knows how to describe a software according to FDF and how does the software work. It inherits from a `Software` abstract composite provided by FDF. This composite offers a `Deployment` server interface which gives elementary deployment procedures for install, configure, start, stop, and uninstall the software. The `Software` composite maintains also the software status through a seven state automaton. Three are permanent states (`UNINSTALLED`, `INSTALLED`, `STARTED`) and four are transitory (`INSTALLING`, `STARTING`, `STOPPING`, `UNINSTALLING`). Software states are used by a `Dependencies` component which is responsible for the management of dependencies between software. For instance, when the `start()` or `install()` procedure is called for a software, the `Dependencies` component delegates the method to all dependent softwares which must be installed or started before. Like the deployment descriptions the software personalities are described with the FDF language. The Fractal Deployment Framework has already many personalities for deploying ActiveBPEL, Ant, CORBA, FDF, Fractal, Geronimo, Glassfish, JADE, JASMINe, JAVA, JBoss, JOnAS, Julia, MySQL, OpenCCM, Orchestra, Oscar, PEtALS, Qemu, Tomcat.

Primitive components

The Fractal Deployment Framework provides a library of runnable and low-level components. Runnable components encapsulate a lot of basic functionalities. Encapsulated functionalities allow for example to upload and download files, remotely set and unset shell variables, execute commands, *etc*. These components are composed sequentially by the `install`, `start`, `stop`, and `uninstall` procedures of software personalities. Low-level deployment components are abstractions for deployment mechanisms to access physical host.

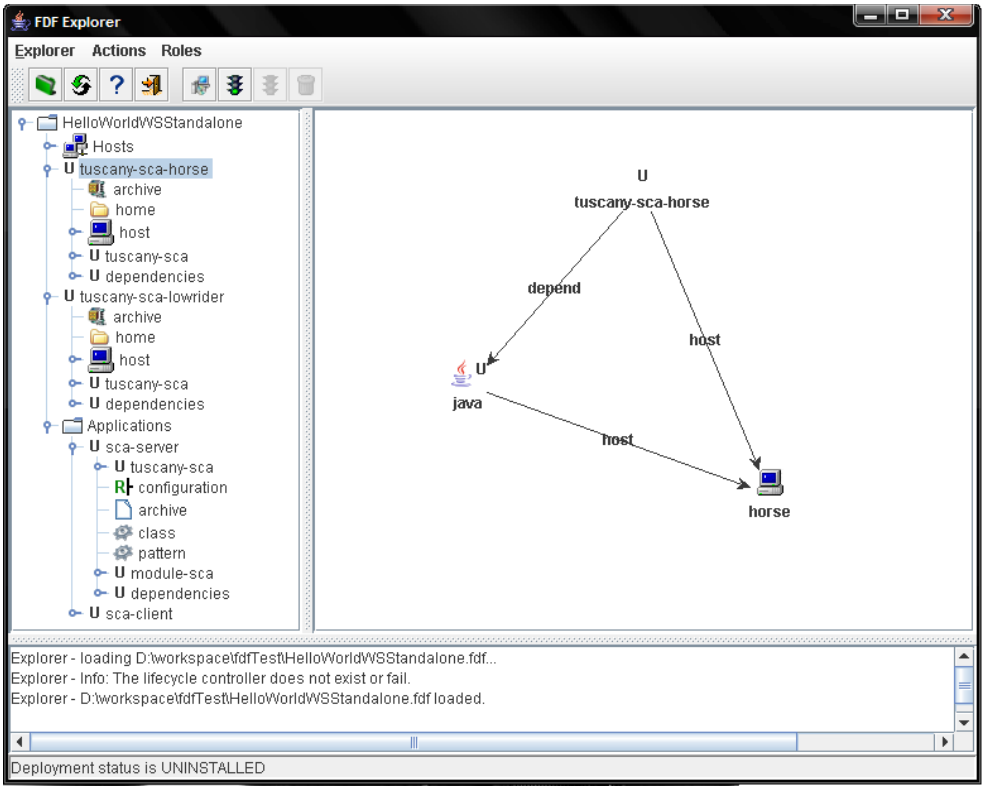


Figure 51: FDF Explorer

Finally, FDF provides also a graphical user interface. This interface, named FDF Explorer (depicted in Figure 51) , enables loading, and browsing of deployment descriptions. Using FDF Explorer, the system administrator can explore system description components and visualize software dependencies. Moreover, the administrator can interact with the explorer in order to install, start, stop, uninstall software described by the loaded deployment file. The explorer also indicates software status.

b) Use case for Service Component Architecture

The following example illustrates how FDF can help system administrator in deployment of SCA applications, SCA runtimes and virtual machines on distributed systems.
Let's consider an application composed by four SCA composites (noted C1, C2, C3, C4). These SCA composites must be installed on different servers accessible on the network (noted M1, M2, M3, M4). According to requirements of services and implementation of SCA composites, the servers need different system configurations. In our case we have one SCA composite developed for the Tomcat container for Tuscany SCA, one for Tuscany SCA standalone, one for a FraSCAti server, and one for FraSCAti running in a PEtALS server. Figure 52 shows the different machines and the software stack required to run these different SCA composites.

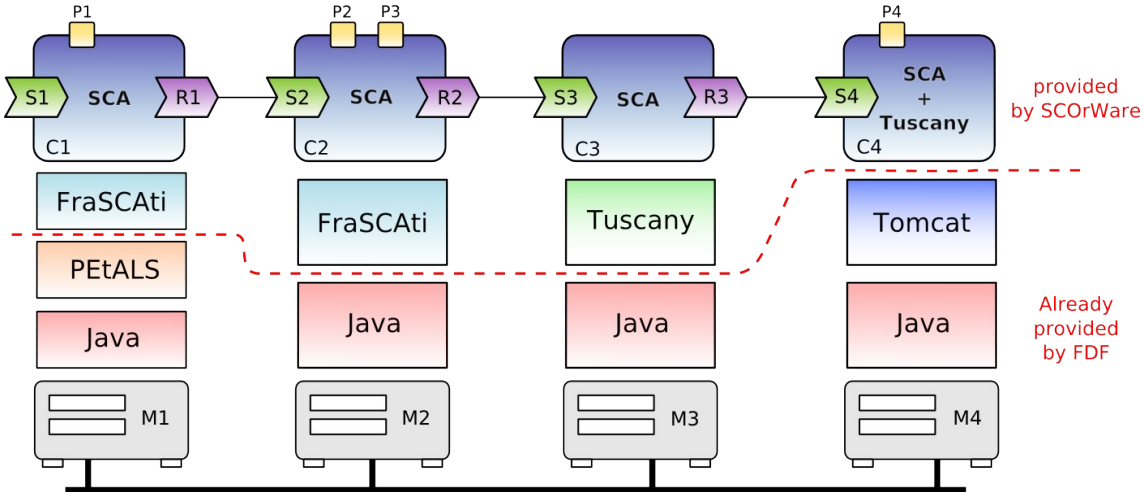


Figure 52: SCA Deployment use case

While manual deployment of a such system architecture can be a boring and repetitive task (install Java virtual machine one by one, for example), FDF allow us to easily install the complete software stack on different machines using a single description. This single description, named FDF deployment file, must first define the system architecture with informations allowing us to remotely access to machines and upload softwares. The following code gives an example definition for the machine M1, M2, M3, M4 :

```
Hosts = INTERNET.NETWORK {

  m1 = INTERNET.HOST {
    hostname = INTERNET.HOSTNAME(m1);
    user     = INTERNET.USER(admin,password,key);
    transfer = TRANSFER.SCP;
    protocol = PROTOCOL.OpenSSH;
    shell    = SHELL.SH;
    software {
        java = JAVA.JRE {
          archive = JAVA.ARCHIVE(/archives/jre.zip);
          home    = JAVA.HOME(/tmp/jre);
        }
      }
    }
  }

  m2 = INTERNET.HOST {
    hostname = INTERNET.HOSTNAME(m2);
    user     = INTERNET.USER(admin,password,key);
    transfer = TRANSFER.SCP;
    protocol = PROTOCOL.OpenSSH;
    shell    = SHELL.SH;
    software {
        java = JAVA.JRE {
          archive = JAVA.ARCHIVE(/archives/jre.zip);
          home    = JAVA.HOME(/tmp/jre);
        }
      }
    }
  }

  m3 = INTERNET.HOST {
    hostname = INTERNET.HOSTNAME(m3);
    user     = INTERNET.USER(admin,password,key);
    transfer = TRANSFER.SCP;
    protocol = PROTOCOL.OpenSSH;
    shell    = SHELL.SH;
    software {
        java = JAVA.JRE {
          archive = JAVA.ARCHIVE(/archives/jre.zip);
          home    = JAVA.HOME(/tmp/jre);
```

```
        }
      }
    }
  }

  m4 = INTERNET.HOST {
    hostname = INTERNET.HOSTNAME(m4);
    user     = INTERNET.USER(admin,password,key);
    transfer = TRANSFER.SCP;
    protocol = PROTOCOL.OpenSSH;
    shell    = SHELL.SH;
    software {
        java = JAVA.JRE {
          archive = JAVA.ARCHIVE(/archives/jre.zip);
          home    = JAVA.HOME(/tmp/jre);
        }
      }
    }
  }
}
```

This code example defines that our system architecture is composed of the machines M1, M2, M3, M4. These machines provide a UNIX shell command and are remotely accessible through the SSH protocol. Moreover it indicates that file transfers can be done through SCP (Secure Copy). Thanks to the JAVA personality, which is already available in the FDF current implementation, we can easily describe deployment of the Java Virtual Machine. Like many software descriptions, we simply indicate to FDF the location of software `archive` and the install directory (`home`). In this example we define that a Java runtime environment is deployed by default on the machines M1, M2, M3, M4.

Deployment of PEtALS and Tomcat servers is done through the PEtALS and Tomcat personalities, also already available in FDF. To describe deployment for PEtALS and Tomcat servers we add the following description :

```
petals-on-M1 = PEtALS.SERVER {
  archive = PEtALS.ARCHIVE(/archives/petals.zip);
  home    = PEtALS.HOME(/tmp/petals);
  host    = Hosts/m1;
}

tomcat-on-M4 = TOMCAT.SERVER {
  archive = TOMCAT.ARCHIVE(/archives/tomcat.zip);
  home    = TOMCAT.HOME(/tmp/tomcat);
  host    = Hosts/m4;
}
```

Like the Java personality we just have to specify where the archives of PEtALS and Tomcat are located and where we want to deploy them. Additionally, it is possible to configure the PEtALS or Tomcat servers using properties. We refer to the FDF user guide for more details on server configuration through properties.

We have described deployment for lower part for the system of Figure 52, with personalities already provided by FDF. We now have to deploy FraSCAti (for machines M1, M2) and Tuscany (for machines M3, M4). To do this we propose to extend FDF with new personalities which enable deployment of SCA based applications and runtimes. In section c) we describe how we integrate the new personalities for SCA, Tuscany, and FraSCAti provided in the context of the SCOrWare project, with the Java, PEtALS and Tomcat personalities available in FDF. In section d) we give specification details for these new personalities and give examples related to the use case presented in Figure 52.

c) Dependencies between personnalities

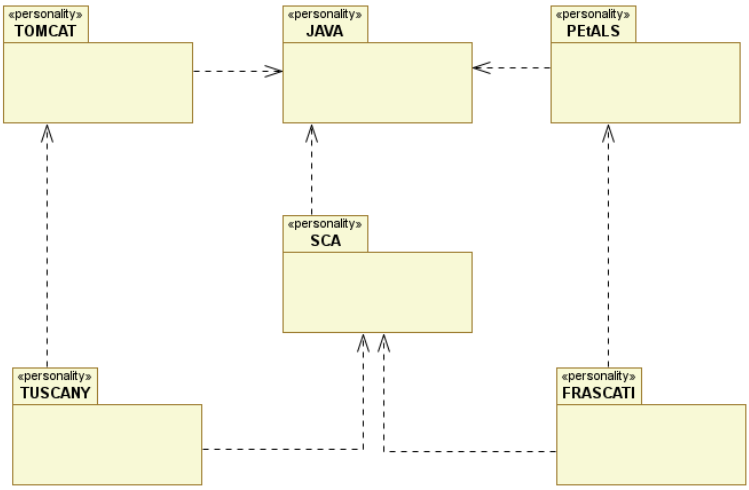


Figure 53: Dependencies between personalities

`SCA`, `TUSCANY`, and `FRASCATI` are defined by FDF personalities. `SCA` is an abstract personality which defines how typical SCA applications and/or runtime are described according to FDF, while the `TUSCANY` and `FRASCATI` personality extends the SCA personality with runtime specific parameters and deployment procedures. Moreover, Tuscany and FraSCAti personalities respectively depends on Tomcat and PEtALS personalities, since they provide containers for these application servers.

Tuscany depends also from the Tomcat personality. The Tuscany personality extends the definition of a web application archive from Tomcat in order to integrate the ability to configure communication protocols and properties. In the Tuscany personality, a web application archive is a configurable SCA domain running on Apache Tomcat. Similarly to Tuscany, FraSCAti depends from the PEtALS personality. Because, FraSCAti can be integrated into a PEtALS server as a JBI component, the FraSCAti personality extends the JBI definition in PEtALS to define how to deploy FraSCAti on PEtALS, and extends the definition of service assembly to describe SCA Domain deployment.

Tomcat, PEtALS, Tuscany, FraSCAti personalities depend also on the Java personality, since these software are implemented into the Java language. Figure 53 illustrates dependencies between personalities in the context of the SCOrWare platform.

d) SCOrWare Personalities

SCA personality

The SCA personality allows to describe, configure and install SCA composites upon SCA runtime. This personality is an abstract and runtime independent description for the deployment of an SCA application. It has to be extended by each SCA runtime which needs to be deployed *via* FDF. Basically, the SCA personality is made of two main components : the `COMPOSITE` software and the `RUNTIME` software. The UML diagram of this FDF personality is given in Figure 54.

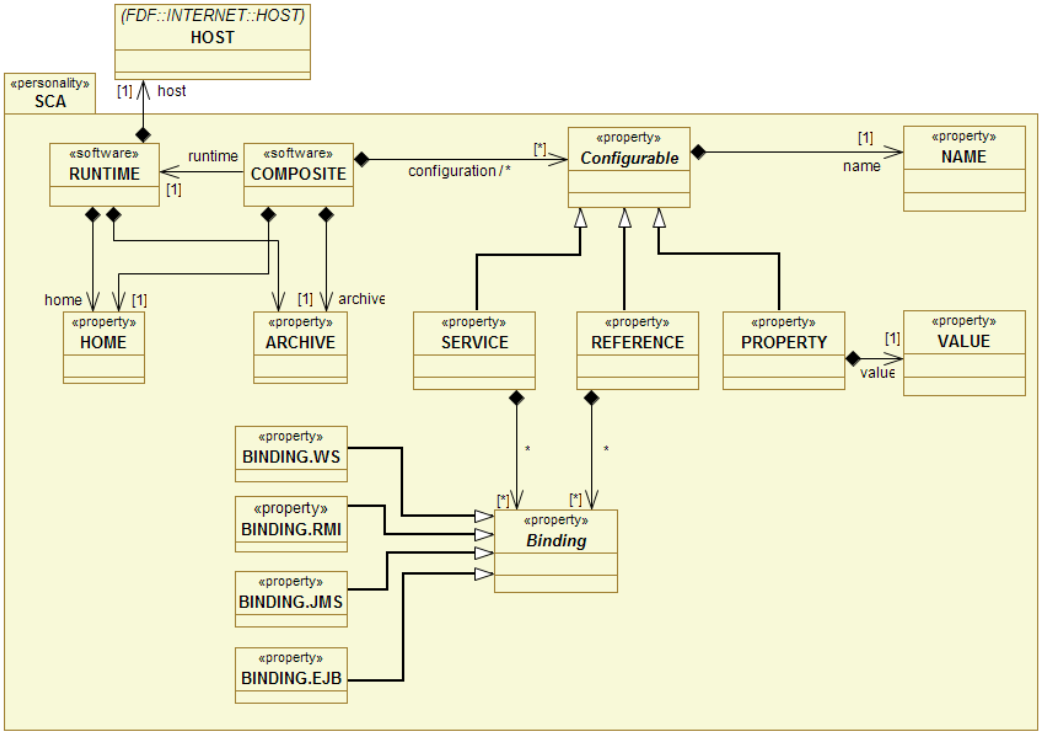


Figure 54: SCA personality diagram

- **SCA Runtime**

The SCA runtime is defined by a component named `SCA.RUNTIME`. It requires three subcomponents defining the SCA runtime archive, the home directory where to unpack it, and the host where the SCA runtime is deployed. Basically, an SCA runtime is described like :

```
sca-runtime = SCA.RUNTIME {
  archive = SCA.ARCHIVE(sca-archive);
  home = SCA.HOME(sca-home-directory);
  host = Hosts/hostname;
}
```

`SCA.RUNTIME` is an abstract description of an SCA runtime. Any personality describing an SCA runtime must extend or refer to the `SCA.RUNTIME` definition. Table 9 lists the subcomponents of the `SCA.RUNTIME` definition.

Name	Type	Description
`archive`	`SCA.ARCHIVE`	Archive of the SCA runtime
`home`	`SCA.HOME`	Target home directory of the SCA runtime
`host`	`INTERNET.HOST`	A host of the physical infrastructure

Table 9: SCA runtime subcomponents

- **SCA Composite**

SCA Composite defines an archive containing one or more composite descriptions and their implementation. The descriptions define SCA components to deploy on one or many hosts. Like other personalities in FDF, the SCA

composite extends the `SOFTWARE` personality. The SCA composite personality defines the application archive and the home directory. An SCA composite also refers to an SCA runtime. The minimal description of an SCA application in FDF is written below :

```
sca-application = SCA.COMPOSITE(archive) {
  runtime = /sca-runtime;
}
```

Table 10 lists the subcomponents of the `SCA.COMPOSITE` definition.

Name	Type	Description
`runtime`	`SCA.RUNTIME`	SCA runtime for the composite

Table 10: SCA composite subcomponent

According to FDF, an SCA business application is defined by a component named `SCA.COMPOSITE` which has one parameter : the path to application archive. In order to run, the SCA application requires a runtime which is described as a sub component of the `SCA.COMPOSITE`.

Any SCA personality is able to introspect application archive in order to configure assembly descriptions. This particularity allows the system administrator to adapt SCA applications with the system architecture. A system administrator, through the SCA personality, can deploy any SCA application without having to rewrite the composite descriptions. This personality provides a way to specify communication protocols to use between composite services and clients, and to fix values of composite properties. For example, if an SCA application provides a service accessible through SCA binding, the system administrator can configure the deployment of this application in order to use the Web service binding (allowing the application to be interoperable with non SCA client). To personalize SCA applications we use an optional `configuration` subcomponent. A description for the deployment with FDF of a configured SCA application is given below :

```
sca-server = SCA.COMPOSITE(archive) {
  runtime = /sca-runtime;

  configuration {
    service = SCA.SERVICE(name) {
     binding = SCA.BINDING.RMI(host, port, name);
    }
    header = SCA.PROPERTY(name,value);
  }
}
```

This sample describes an application configured to use Java RMI as binding protocol for the service `name`. The configuration of the Java RMI binding requires also to specify specific protocol parameters. For this example with Java RMI, we have to indicate the `port`, the `hostname` and the `name_in_registry`. Needed parameters depend on the type of protocols we want to use. These parameters correspond to attributes defined for each binding protocol in the SCA assembly model specification. We give FDF keywords to refer to protocols and their parameters in the following tables :

Definition	Protocol
`SCA.BINDING.RMI`	Java Remote Method Invocation using optional parameters
`SCA.BINDING.RMI(host,port,name)`	Java Remote Method Invocation with `host, port, name` parameters

Parameter	Type	Description
`port`	`SCA.BINDING.RMI.PORT`	Value of a TCP/IP port of the RMI registry
`host`	`SCA.BINDING.RMI.HOST`	Host name for the RMI registry
`name`	`SCA.BINDING.RMI.NAME`	Name of the service in the RMI registry

Table 11: RMI binding and parameters

Definition	Protocol
`SCA.BINDING.WS`	Simple Object Access Protocol using optional parameters
`SCA.BINDING.WS(element,location,endpoint,uri)`	Simple Object Access Protocol using optional parameters `element` and `location` parameters

Parameter	Type	Description
`endpoint`	`SCA.BINDING.WS.ENDPOINT`	Endpoint of a service or a reference
`element`	`SCA.BINDING.WS.WSDL-ELEMENT`	Specify URI of a WSDL element
`location`	`SCA.BINDING.WS.LOCATION`	Location of the WSDL document
`uri`	`SCA.BINDING.WS.URI`	Explicit endpoint URI

Table 12: Web Service binding and parameters

Alias	Protocol
`SCA.BINDING.JMS`	Java Messaging Service using optional parameters
`SCA.BINDING.JMS(uri)`	Java Messaging Service using `uri` parameter

Parameter	Type	Description
`uri`	`SCA.BINDING.JMS.URI`	URI for connection type and informations

Table 13: JMS binding and parameters

Let's note that the sample configuration describes configuration for one service of an SCA domain, while FDF is able to configure several services and references at once. FDF enables also to set properties of the composite. Setting composite is done using the `SCA.PROPERTY` component. It takes as parameters the name of the property and the value to set. Like protocol configuration, we can set multiple properties at once.

The former example is a sample FDF description for configuring an SCA domain. FDF doesn't differentiate processing of composite service and composite reference when it introspect an SCA archive and configures the assembly description. This means that a configuration can be defined once and, thank to FDF, reused for other SCA archive. This feature is quite useful when the administrator want to configure the binding protocol used between an SCA client and an SCA server. For instance if we wants to configure a client application using the configuration of the last example, we simply write :

```
sca-client = SCA.COMPOSITE(archive) {
  runtime = /sca-runtime;

  configuration {
    reference = SCA.REFERENCE(name) {
      binding = sca-server/configuration/service;
    }
  }
}
```

Tuscany personality

The Tuscany personality defines how to deploy an SCA application based on a Tuscany runtime. This personality is based on the abstract SCA personality which gives a basic definition for both SCA composite and runtime. The Figure 55 gives an overview of this personality and its definitions.

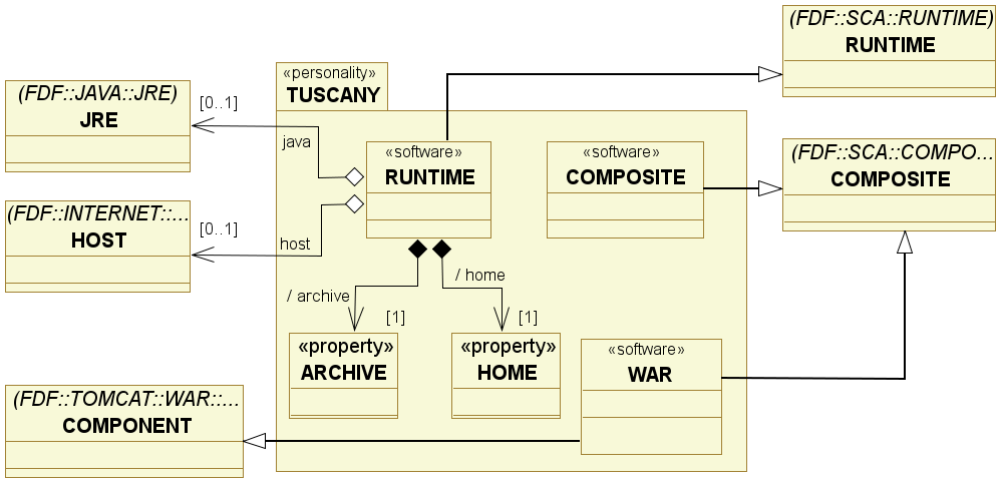


Figure 55: Tuscany personality diagram

The Tuscany personality contains the `TUSCANY.COMPOSITE`, the `TUSCANY.WAR` and the `TUSCANY.RUNTIME` definitions. They respectively define deployment of (a) an SCA application on the Tuscany runtime, (b) an SCA application using Tuscany in a Tomcat container, and (c) Tuscany as a standalone runtime.

- **Tuscany Standalone Runtime**

The Tuscany personality allows to deploy the standalone Java version of Apache Tuscany SCA runtime. From FDF internal point of view, Tuscany is defined as an installable software depending on the Java runtime environment (see the FDF documentation for more details about the Java personality). However, from administrator point of view, the Tuscany runtime is a component named `TUSCANY.RUNTIME` which requires three components defining the Tuscany archive, the home directory where to unpack it, and the host on which Tuscany must be deployed. According to the example described in Figure 52, the following code define deployment of Apache Tuscany SCA runtime on machine M3.

```
tuscany = TUSCANY.RUNTIME {
  archive = TUSCANY.ARCHIVE(/archives/tuscany.zip);
  home = TUSCANY.HOME(/tmp/tuscany);
  host = Hosts/m3;
}
```

Let's note that the Tuscany runtime is never started through shell commands. It is invoked by SCA application main classes. As a consequence, Tuscany will never be started directly *via* the FDF Explorer.

- **SCA Composite on Tuscany Standalone Runtime**

The Tuscany composite defines an SCA application which is designed to be deployed on the Apache Tuscany SCA standalone runtime. Typically, an SCA application developed to run on Tuscany runtime provides a main class which bootstraps the runtime with an assembly description. The code executed by the SCA application main class is :

```
SCADomain scaDomain = SCADomain.newInstance(composite-file);
```

where `composite-file` parameter indicates the *.composite* file of the assembly description to run.

Thus the FDF definition of an SCA domain executed upon the Tuscany runtime must specify the Java class with the bootstrap code. The `TUSCANY.COMPOSITE` extends the `SCA.COMPOSITE` with the bootstrap class parameter. The next example gives description for deployment of an SCA composite on the machine running the Apache Tuscany SCA standalone runtime (to remind, machine M3).

```
C3 = TUSCANY.COMPOSITE(class, archive) {
  runtime = /tuscany;
}
```

However, an SCA composite can be configured during deployment process, meaning binding protocols and properties can be set as described for the SCA personality. The next example shows how to configure the service S3 for the composite noted C3 in Figure 52.

```
C3 = TUSCANY.COMPOSITE(Main, c3.zip) {
  runtime = /tuscany;

  configuration {
    service = SCA.SERVICE(S3) {
     binding = SCA.BINDING.RMI(m3, 1099, c3s3);
    }
    reference = SCA.REFERENCE(R3) {
      binding = C4/configuration/service;
    }
  }
}
```

- **SCA Composite on Tomcat container**

The Tuscany runtime provides also support for running into a web container of Apache Tomcat. In this case the SCA application is packaged as a Web Application Resource file (WAR) which contains both the SCA application and the Tuscany runtime (executed as a servlet). According to FDF's terminology, we define a Tomcat container for Tuscany with the keyword `TUSCANY.WAR`. Actually, it extends the existing `TOMCAT.WAR` definition provided by the Tomcat personality (see the FDF user guide for more details about the Apache Tomcat personality). The code below permits to deploy a WAR packaged Tuscany SCA application on the Tomcat server previously described (see Figure 52)

```
C4  = TUSCANY.WAR {
  archive = TOMCAT.WAR.ARCHIVE(c4.war);
  name = TOMCAT.WAR.NAME(c4);
  tomcat = tomcat-on-M4;
}
```

Where `archive` and `name` refers to the Tomcat WAR file and the Web Application context while `tomcat` refers to the description of a Tomcat server. The `TUSCANY.WAR` definition inherits from the `SCA.DOMAIN`. Thus, WAR packages can be introspected in order to configure binding protocols and properties, when they are deployed on a Tomcat server. For instance, applying the sample configuration described for `SCA.COMPOSITE` on component C4 (see Figure 52) will result in :

```
C4 = TUSCANY.WAR {
  archive = TOMCAT.WAR.ARCHIVE(c4.war);
  name = TOMCAT.WAR.NAME(c4);
  tomcat  = tomcat-on-M4;

  configuration {
    service = SCA.SERVICE(S4) {
     binding = SCA.BINDING.RMI(m4, 1099, c4s4);
    }
    P4 = SCA.PROPERTY(p4,value);
```

```
  }
}
```

FraSCAti personality

The FraSCAti personality allows to deploy the FraSCAti SCA runtime provided by the SCOrWare project. Like Tuscany, it extends the SCA personality. Figure 56 gives the UML model for FraSCAti personality.

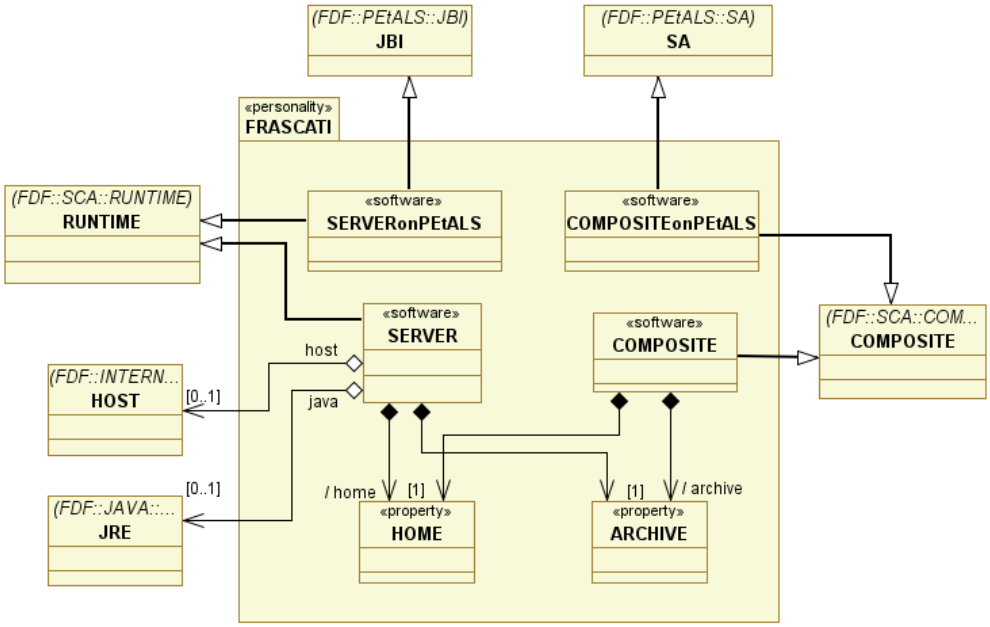


Figure 56: FraSCAti personality diagram

- **FraSCAti Runtime**

The FraSCAti runtime is indicated by `FRASCATI.SERVER`, it extends the `SOFTWARE` definition and thus require to describe the archive location, the home directory and the host where FraSCAti must be deployed. The code below describes deployment of FraSCAti according to the use case.

```
frascati = FRASCATI.SERVER {
  archive = FRASCATI.ARCHIVE(/archives/frascati.zip));
  home = FRASCATI.HOME(/tmp/frascati);
  host = Hosts/m2;
}
```

The FraSCAti runtime is a server application. It requires to be started before uploading SCA applications. Dependency between SCA applications and FraSCAti is automatically managed by FDF. When the administrator deploys an SCA application on a host, FDF automatically starts (or deploys and starts) the FraSCAti server. In opposite, when the last SCA application is uninstalled, FDF automatically stops the FraSCAti server.

- **SCA Composite on FraSCAti**

The FraSCAti Composite defines an SCA application (an archive containing composite descriptions and their implementation) to deploy on the FraSCAti runtime server. Like Tuscany, FraSCAti is a standalone runtime which provides support for running SCA composites. But, while Tuscany is bootstrapped by the application, FrasSCAti runs as an SCA application server, waiting for SCA composites to deploy. The main composite to start when uploading an SCA application to the FraSCAti server, is defined into a configuration file. This configuration file describes additional parameters required for deploying an SCA application on FraSCAti. Consequently, an SCA composite implemented for

running with FraSCAti is defined with the keyword `FRASCATI.COMPOSITE` and an `archive` as parameter. Moreover, as the `FRASCATI.COMPOSITE` component extends the `SCA.COMPOSITE` , SCA application can again be configured while they are deployed. This is an example of an SCA application deployed on FraSCAti and configured.

```
C2 = FRASCATI.COMPOSITE(c2.zip) {
  runtime = /frascati;

  configuration {
    service = SCA.SERVICE(S2) {
      binding = SCA.BINDING.RMI(m2, 1099, c2s2);
    }
    reference = SCA.REFERENCE(R2) {
      binding = C3/configuration/service;
    }
    P2 = SCA.PROPERTY(p2,value);
    P3 = SCA.PROPERTY(p3,value);
  }
}
```

- **FraSCAti on PEtALS server**

Description for PEtALS server deployment is already provided by FDF (see the FDF documentation). We can directly use the PEtALS personality to describe our server. The following PEtALS deployment description reminds example presented in Figure 52 :

```
petals-on-M1 = PEtALS.SERVER {
  archive = PEtALS.ARCHIVE(/archives/petals.zip);
  home    = PEtALS.HOME(/tmp/petals);
  host    = Hosts/m1;
}
```

While the code above describes a PEtALS server to install on a host, the node is not able to run directly SCA applications. To support SCA application the FraSCAti runtime must be deployed upon the PEtALS node. In the context of the SCOrWare project, the FraSCAti runtime can be embedded as a JBI component in order to execute SCA applications on PEtALS and take benefits from services already existing. The FraSCAti component is indicated by `FRASCATI.SERVERonPEtALS`. It extends the JBI component noted `PEtALS.JBI` in the PEtALS personality

```
frascati-jbi = FRASCATI.SERVERonPEtALS {
  archive = PEtALS.ARCHIVE(/archives/frascati-jbi.zip);
  petals  = /petals-on-M1;
}
```

- **SCA applications on FraSCAti PEtALS**

PEtALS is a Entreprise Service Bus (ESB) which implements the Java Business Integration specification. Like an SCA application designed to run on Tuscany in a Tomcat container, the FraSCAti runtime provides support for running into PEtALS servers. The SCA application is indicated by `FRASCATI.COMPOSITEonPEtALS`, it extends the description of a service assembly in the PEtALS personality and allows to configure the SCA application when it is deployed. For instance, an SCA composite running on a PEtALS server is described by :

```
C1 = FRASCATI.COMPOSITEonPEtALS {
  archive = PEtALS.ARCHIVE(/archives/c1.zip);

  configuration {
    reference = SCA.REFERENCE(R1) {
     binding = C2/configuration/service;
    }
    P1 = SCA.PROPERTY(p1,value);
  }
}
```

8.2.3 Implementation

This part gives implementation details of the Tuscany and FraSCAti personalities for FDF. Since FDF provides several primitive components, most of the implementation of the SCA, Tuscany and FraSCAti personalities reuse existing FDF components. For instance, we can reuse primitive components allowing to transfer, extract files, execute shell commands, etc... Moreover, the SCA based personality allows to configure the SCA applications and their bindings while they are installed on the SCA runtime. In order to provide this feature, we have extended FDF with a primitive component responsible for executing XSL transformation [29].
In the following paragraphs, we will focus on the FraSCAti personality and the component allowing to configure SCA composites. We will not describe Tuscany personality since it is very close to FraSCAti. First we give an overview of the FraSCAti personality with the procedures which allow FDF to deploy an SCA runtime and applications. Then we will focus on the implementation of the component responsible for processing composite documents.

Personality overview

The FraSCAti and Tuscany personalities reuse lot of existing FDF implementation. Rather than giving details on the FDF implementation, this part describes how to create an FDF personality using the construct already available. Moreover, although FDF provides a specific language for describing software deployment, we concentrate more on deployment concepts introduced into FDF than the language syntax. (We prefer to refer to the FDF documentation for more details).
Generally, implementing a new FDF personality consists in adding a new software definition for FDF. This can be done by extending the component named "Software" which allows to describe a software according to FDF. For instance, in order to support FraSCAti with FDF, we have added two software definition which describe the FraSCAti runtime and and an SCA application running with FraSCAti.

- **Runtime**

The FraSCAti runtime personality allows FDF to deploy the FraSCAti runtime platform. It provides procedures which allow FDF to install, start, stop, or uninstall the FraSCAti platform. These procedures wrap the deployment mechanism described by the end user documentation of FraSCAti. Actually, the FraSCAti runtime personality runtime use the FDF language to define the command sequence for installing, starting, stopping or uninstalling the SCA platform.
The following figure (57) shows part of the FraSCAti personality defining the runtime:

```
1  FRASCATI.SERVER
2  = JAVA.DependOn(frascati),
3    software.Installable(frascati,Frascati,archive,home),
4    software.Iconable(frascati,FRASCATI/SERVER.png)
5   {
6
7     # archive is required.
8     archive = FRASCATI.ARCHIVE(UNDEFINED);
9
10    # home is required.
11    home = FRASCATI.HOME(UNDEFINED);
12
13    frascati {
14
15      internal-deployment {
16
16        configure {
17          set-home = SHELL.SetVariable(FRASCATI_HOME,#[home]);
18          add-path = SHELL.AddPath(#[home]/bin);
19        }
20
```

Figure 57: FraSCAti runtime deployment

Actually the FraSCAti runtime is started for each SCA application , it does not execute as an application server (Like Apache Tomcat server for instance). As a consequence, the FDF procedures which allow to start or stop FraSCAti (line 21 and 26) does nothing. However, once installed, FraSCAti needs to be configured. Then the "configure" step sets the FraSCAti home environment variable and adds its command to the commands path.
The FraSCAti personality imports procedures given by the definition of a installable software (line 3). This generic

definition gives install/uninstall procedures for applications packaged as an archive. The archive and home parameters respectively defined at lines 8 and 11 provide path to the FraSCAti archive and the directory where this archive must be unpacked on the targeted host. Finally, FraSCAti needs Java in order to run. With FDF we can express such dependencies. Then, when deploying an application, FDF will recursively process software on which the current software depends. The dependency on Java in the FraSCAti personality is expressed at line 2 with the definition named "JAVA.DependOn".

- **Composite**

The FraSCAti personality also provides a FDF definition for describing an SCA application to deploy on the FraSCAti runtime. In this case, this definition wraps mechanism given by the FraSCAti runtime to run an SCA application. From FDF point of view, an SCA application is a software which depends on the FraSCAti runtime (see figure 58). This definition also provides methods for installing / uninstalling SCA applications on top of FraSCAti and start / stop those applications.

```
 1  FRASCATI.COMPOSITE(name)
 2  =org.objectweb.fdf.components.software.Software(composite, Frascati Composite)
 3  , FRASCATI.DependOn(composite)
 4  , software.ShareHost(frascati)
 5  {
 6   # archive is required.
 7   archive = FRASCATI.COMPOSITE.ARCHIVE(UNDEFINED);
 8
 9   name = FRASCATI.COMPOSITE.NAME(${name});
10
11   # The composite's home is arbitrarily chosen as FRASCATI_HOME
12   home= /frascati/internal-parameters;
13
14  /**
15   * Frascati composite implementation
16   */
17   composite {
18
19     internal-deployment {
20
21       install {
22         upload-archive = TRANSFER.Upload(#[archive],#[home]/#[archive-filename]);
23       }
24
....
30
31       start {
32         run = SHELL.ForkScript(frascati,#[name] #[home]/#[archive-filename]);
33       }
34
....
39
40        uninstall {
41          delete-archive = SHELL.RemoveFile(#[home]/#[archive-filename]);
42       }
43    }
44 }
```

Figure 58: FraSCAti composite deployment

The current FraSCAti distribution uses SCA applications packaged into JAR file. Then the install procedure (line 21) has to upload the application archive on the host where FraSCAti has been deployed. This operation is provided by the procedure named "TRANSFER.Upload" provided by FDF. In opposite, the uninstall procedure removes the archive of the SCA application. To enable FDF to start the SCA application using FraSCAti, the start procedure uses shell commands to call the FraSCAti startup script with appropriate parameters. However, there's actually nothing to do to stop FraSCAti.
The FDF definition for SCA application also provides the ability to configure the SCA application bindings and properties when the archive is uploaded (not showed in figure 58). In the next section we details the XSL transform engine which allows to configure SCA composites. This engine has been added to implementation of FDF in order to support transformation for any kind of XML documents.

XML transformation for FDF

SCA composites are described through an XML syntax. As already seen in part 8.1.3, XML documents can be processed with several XML tools, and we have chosen to use The Eclipse modeling framework into the SCA assembly factory for processing SCA composites.
When deploying SCA applications, the Fractal Deployment Framework is able to configure SCA composites, more specifically its properties and bindings. This allows to adapt the deployed SCA application with the underlying network architecture. However we don't reuse the components implemented into the SCA assembly factory. We have chosen to use an XSL transformation engine. XSL transformation is enough flexible to process any kind of XML documents with FDF, contrary to the assembly factory parser which is dedicated to SCA syntax. XSL transformation allows to process SCA composite files to change bindings or properties as well as it can help in processing different XML documents with other FDF personalities.
The XML transformation process is implemented as a couple of primitive components into FDF : a component responsible for XML transformation and a filter component representing a transformation to apply.

- **Transform Engine**

The transform engine implemented into FDF is based on the Apache Xalan XSLT processor, it uses XSL to transform XML documents. The engine also uses TrueZip API to access JAR/Zip archive content without having to extract files.
The transform component implements a pipe for XML transformations. The pipe allows to chain several transformations to apply on a XML document. This allows FDF to process several XML documents without having to rebuild the pipe each time. The transformation rules performed by the engine are defined using transform filter components. The figure 59 gives an overview of the piped transformation.

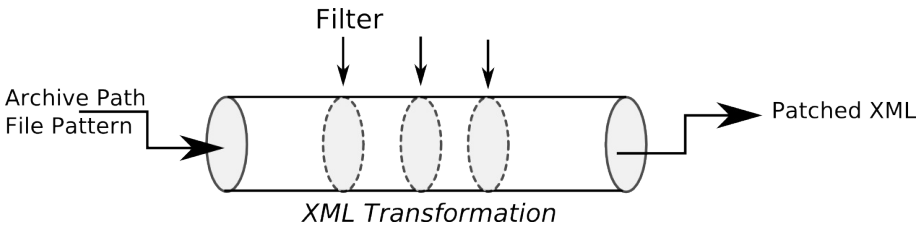


Figure 59: Piped XML transformation

Since the transform engine is implemented as a FDF primitive component, it provides a runnable interface allowing FDF to start the transform process when deploying SCA applications. The transform component is also configured with two parameters : the absolute path to the SCA application archive containing composite definition and a regular expression which permits to filter files processed by the engine. The path to the SCA application is provided by the FDF end user; The regular expression for filtering files is fixed in the software personality.

The engine component also requires filter components which define transformations to apply. When building the pipe for XML transformations, the transform component iterates over filter components connected to its client interface. The parameters provided by filter components are used to chain the XSL transformations.

- **Transform Filter**

Filter components provide parameters used by the XSL transform engine. A filter component allows the transform engine to retrieve the XSL document associated to an XML filter and its parameters. Filter component is configured with end user definition. For instance, when defining settings in the FraSCAti personality, the user effectively describes part of the FDF architecture which is still unknown. The settings defined by the user allows to configure the FDF architecture with a set of filter components reflecting the transformations to apply on SCA composite files. The user can easily defines many filters to add to the transformation process, while the framework is responsible for creating filter components from their FDF definitions, and linking them with the engine component.

To sum up , we have principally added to the Fractal Deployment Framework the ability to process and transform XML documents. We have defined a new primitive component which is able to process SCA composites (but not limited to) according to the list of settings provided by FDF users. Combined with the already available primitive components for installing, and starting software; we have easily added to FDF new personalities supporting SCA applications, the

Tuscany and the FraSCAti runtimes.

8.2.4 Summary

The Service Component Architecture specification defines a programming model for development of distributed applications, implemented into reusable building blocks, with different languages and multiple communication protocols. However, SCA does not deal with deployment of applications or SCA runtimes. In this part we have proposed to address deployment of both SCA applications and runtimes with the Fractal Deployment Framework. FDF is able to manage deployment for any kind of software, from the operating system to the end-user application, on heterogeneous and distributed systems. FDF is composed of personalities which define deployment for a specific software. Currently, personalities for Java, Apache Tomcat Server, PEtALS server are already available. In the context of the SCOrWare project we have specified tree new personalities to define deployment of SCA applications, Tuscany and FraSCAti runtimes. While SCA personalities describe an application or a runtime to deploy, they also permit to configure bindings and properties of SCA applications during deployment process. As a result, FDF allows to deploy and adapt easily the SCA applications to the targeted physical infrastructure. It avoids to the system administrator to modify manually the SCA application. The system administrator must only define its system infrastructure and the software to deploy, while FDF is in charge of deploying, configuring, starting, stopping and removing the set of software.

8.3 Autonomic Support

8.3.1 Component-based management

Tune implements a component-based management approach. Component-based management aims at providing a uniform view of a software environment composed of different types of servers. Each managed server is encapsulated into a component and the software environment is abstracted as a component architecture. Therefore, deploying, configuring and reconfiguring the software environment is achieved by using the tools associated with the used component-based middleware. Any software managed with Tune is wrapped into a Fractal component which interfaces its administration procedures. Therefore, the Fractal component model is used to implement a management layer (Figure 1) on top of the legacy layer (composed of the actual managed software).

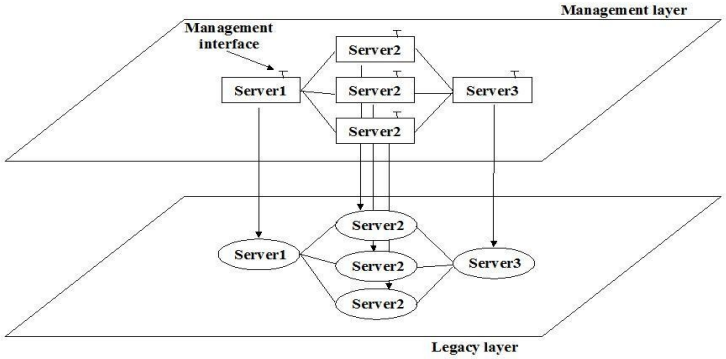


Figure 60: management layer for a software organization

In a first prototype (Jade, a predecessor of Tune), the implementation of management policies was directly relying on the interfaces of the Fractal component model:

- A wrapper was implemented as a Fractal component, developed in Java, which main role is to reflect management/control operations onto the legacy software. Examples of such management operations are the assignment of the component attribute which is reflected by the wrapper in the legacy software configuration (generally a configuration file), or setting up a binding between a client and a server component which is reflected by the wrapper at the legacy layer by binding the two associated legacy software using a specific binding protocol.
- the description of a software architecture to be deployed was described in a Fractal ADL file. This ADL file describes in an XML syntax the set of components (wrappers) to instantiate (which will in turn deploy the associated legacy software components), their bindings and their configuration attributes.

- reconfigurations were developed in Java, relying on Fractal APIs. These APIs allow invoking components' management interfaces or Fractal control interfaces for assigning components' attributes, adding/removing components and updating bindings between components.

8.3.2 Management policy specification

Component-based autonomic computing has proved to be a very convenient approach. But as our system was used by external users (external to our group), we rapidly observed that the interfaces of a component model are too low-level and difficult to use. In order to implement wrappers (to encapsulate existing software), to describe deployed architectures and to implement reconfiguration programs, the administrator of the environment has to learn (yet) another framework, the Fractal component model in our case.
More precisely, our previous experiments showed us that:

- wrapping components is difficult to implement. The developer needs to have a good understanding of the component model we use (Fractal). Regarding wrapping, our approach 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 an equivalent wrapper. Therefore, an administrator doesn't have to program any implementation of Fractal component.
- architectures are not very easy to describe. ADLs are generally very verbose and still require a good understanding of the underlying component model. Moreover, if we consider large scale software infrastructure such as those deployed over a grid, describing an architecture composed of a thousand of servers requires an ADL description file of several thousands of lines. Our approach is to reuse UML formalisms for graphically describing architecture schemas. First, a UML based graphical description of such an architecture is much more intuitive than an ADL specification, as it doesn't require expertise of the underlying component model. Second, the introduced architecture schema is more abstract than the previous ADL specification, as it describes the general organization of the application to deploy (types of software, interconnection pattern) in intension, instead of describing in extension all the software instances that may compose the architecture. This is particularly interesting for large-scale applications where thousands of servers have to be deployed.
- autonomic managers (reconfiguration policies) are difficult to implement as they have to be programmed using the management and control interfaces of the management layer. This also requires a strong expertise regarding the used component model. Our approach is to reuse UML State Diagrams to define workflows of operations that have to be performed for reconfiguring the managed environment. One of the main advantage of this approach, besides simplicity, is that state diagrams manipulate the entities described in the deployment schema and reconfigurations can only produce a concrete architecture which conforms to the abstract schema, thus enforcing reconfiguration correctness.

a) UML-based formalism for architecture schemas

We adapted the UML class diagram formalism in order to allow specification of architecture schemas, as illustrated in Figure 61 where such a schema is defined for a multi-server organization. An architecture schema describes the overall organization of a software infrastructure to be deployed. At deployment time, the schema is interpreted to deploy a component architecture. Each element (the boxes) corresponds to a software which can be instantiated in several component replicas. A link between two elements generates bindings between the components instantiated from these elements. Each binding between two components is bi-directional (actually implemented by 2 bindings in opposite directions), which allows navigation in the component architecture in order to fetch any configuration attribute of the software infrastructure.

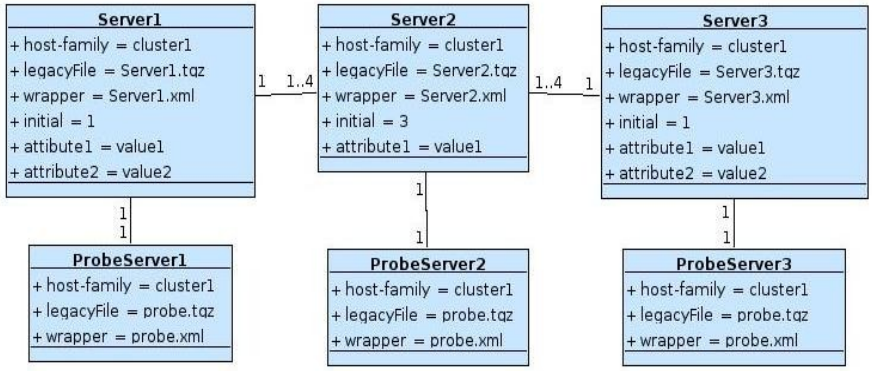


Figure 61: An architecture schema

Figure 2: An architecture schema

An element includes a set of configuration attributes for the software. Most of these attributes are specific to the software, but few attributes are predefined by Tune and used for deployment:

- **wrapper** gives the name of the WDL description of the wrapper,
- **legacyFile** gives the archive which contains the legacy software binaries and configuration files,
- **hostFamily** gives a hint regarding the dynamic allocation of the nodes where the software should be deployed,
- **initial** gives the number of instances which should be deployed.
- **nodeName** gives the identity of the node where the legacy software is deployed. This attribute is implicitly managed by Tune.

In the schema in Figure 61, a cardinality is associated with each link. It constrains the interconnection of the deployed components. An intensional schema may be ambiguous, i.e. the actual deployed component architecture (bindings between components) will depend on the implemented deployment runtime. However, the user may describe a more extensional schema which will better fit his requirements.
The schema in Figure 61 deploys a component architecture as illustrated in Figure 60.

b) A Wrapping Description Language

Upon deployment, the above schema is parsed and for each element, a number of Fractal components are created. These components implement the wrappers for the deployed software, which provide control over the software. Each wrapper component is an instance of a generic wrapper which is actually an interpreter of a WDL specification. A WDL description defines a set of methods that can be invoked to configure or reconfigure the wrapped software. The workflow of methods that have to be invoked in order to configure and reconfigure the overall software environment is defined thanks to a formalism introduced further in this section.
Generally, a WDL specification (illustrated in Figure 62) provides *start* and *stop* operations for controlling the activity of the software, and a *configure* operation for reflecting the values of the attributes (defined in the UML architecture schema) in the configuration files of the software. Notice that the values of these attributes can be modified dynamically. Other operations can be defined according to the specific management requirements of the wrapped software, these methods being implemented in Java.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<wrapper name='Server1'>
      <method           name="start"           class="wrapper.util.GenericStart"
method="start_with_linux" >
      <param ... /> <param ... /> </method>

      <method      name="configure"      class="wrapper.util.ConfigurePlainText"
method="configure">
      <param ... /> <param ... /> </method>

      <method      name="addSlaves"      class="wrapper.util.ConfigurePlainText"
method="configure">
      <param name="config-file" value="conf/Server1.properties" />
      <param name="slave-list" value="Server2.nodeName" /> </method>

      <method         name="stop"         class="appli.wrapper.util.GenericStop"
method="stop_with_linux" >
      <param ... /> <param ... /> </method>
</wrapper>
```

Figure 62: WDL specification

The main motivation for the introduction of WDL are (i) to hide the complexity of the underlying component model (Fractal) and (ii) that most of the needs should be met with a finite set of generic Java methods implementations (that can be therefore reused). We covered the needs of our usecases with very few methods, plaintext and XML file accessors and a shell command launcher. A method definition includes the description of the parameters that should be passed when the method is invoked. These parameters may be String constants, attribute values or combinaison of both (String expressions). All the attributes defined in the architecture schema can be used to pass the configured attributes as parameters of the method invocations.
It is sometimes necessary to navigate in the deployed component architecture in order to configure the software. For instance, in the server organization of Figure 60, at the legacy layer level, Server1 must be configured with the list of nodes where the slave servers (Server2) it is bound with have been launched. Therefore in Figure 62, the *addSlaves* method in the Server1 wrapper must receive this list of nodes in order to configure the Server1 legacy software. The syntax of the method parameters in the wrapper allows navigating in the management layer in order to access the component attributes (in this case the *nodeName* attribute of each Server2), following the bindings between the Server1 component and the Server2 components. *Server2.nodeName* returns the list of *nodeName* attributes of the Server2 components which are bound with the current Server1 component.

c) UML-based formalism for (re)configuration procedures

Reconfigurations are triggered by events. An event can be generated by a specific monitoring component (e.g. probes in the architecture schema) or by a wrapped legacy software which already includes its own monitoring functions. Whenever a wrapper component is instantiated, a communication pipe is created (typically a Unix pipe) that can be used by the wrapped legacy software to generate an event, following a specified syntax which allows for parameter passing. Notice that the use of pipes allows any software (implemented in any language environment such as Java or C++) to generate events. An event generated in the pipe associated with the wrapper is transmitted to the administration node where it can trigger the execution of reconfiguration programs (in our current prototype, the administration code, which initiates deployment and reconfiguration, is executed on one administration node, while the administrated software is managed on distributed hosts). An event is defined as an event type, the name of the component which generated the event and eventually an argument (all of type String).
For the definition of reactions to events, we reused the UML state diagrams formalism which allows specifying reconfiguration. Such a state diagram defines the workflow of operations that must be applied in reaction to an event. An operation in a state diagram can assign an attribute or a set of attributes of components, or invokes a method or a set of methods of components. To designate the components on which the operations should be performed, the syntax of the operations in the state diagrams allows navigation in the component architecture, similarly to the wrapping language.
For example, let's consider the diagram in Figure 63 (on the top) which is the reaction to a Server2 (software) failure. The event (*fixServer2*) is generated by a *ProbeServer2* component instance, therefore the *this* variable references this

ProbeServer2 component instance. Then:

- **this.stop** will invoke the *stop* method on the probing component (to prevent the generation of multiple events),
- **this.Server2.start** will invoke the *start* method on the Server2 component instance linked with the probe. This is the actual repair of the faulting Server2 server,
- **this.start** will restart the probe associated with the Server2.

Notice that state diagram's operations are expressed using the elements defined in the architecture schema, and are applied on the actually deployed component architecture.
The current version of Tune also provides operations which re-deploy components (change location or add component instances) while enforcing the defined abstract architecture schema.

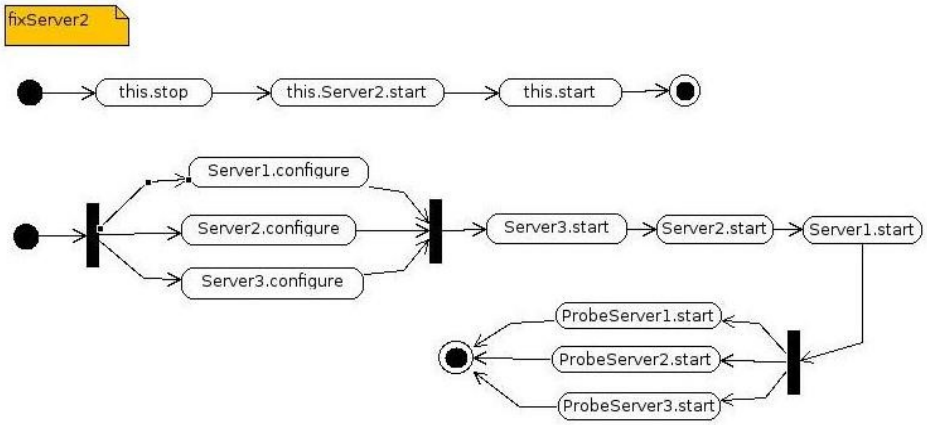


Figure 63: State diagrams for repair and start

A particular diagram is used to start the deployed software, as illustrated in Figure 63 (on the bottom). In this diagram, when an expression starts with the name of an element in the architecture schema (Server1 or Server2 ...), the semantics is to consider all the instances of the element, which may result in multiple method invocations. The starting diagram ensures that (1) legacy configurations must be performed, then (2) the servers must be started following the order Server3, Server2 and Server1. For each type of server, the server is started before its probe.

8.3.3 DSML-Based Autonomic Computing Policies Specification

Our first experiments with Tune focused on the use of XML and UML to take advantage of well-known paradigms and of many existing open source tools. We used the UML2.0 graphical editors provided by the Topcased Eclipse-based toolkit for the description of architectures and reconfiguration diagrams. However, the use of this unified language led us to specialize (pragmatically but sometimes awkwardly) its initial semantics in order to adapt it according to our needs. Because it is difficult to take into account this semantics specialization at the tools level, the user is let with all the freedom offered by UML. For this reason, we are currently studying the possibility to define a dedicated metamodel for management policies definition. This allows us to define a constrained abstract syntax and a dedicated concrete syntax, relying on generic or generative tools, such as TCS or Syntaks for textual formalisms and Topcased or GMF for graphical formalisms. For each point of view that we have taken into account in Tune, we present the corresponding metamodel, offering a constrained, domain-specific and user-friendly languages.

d) *The Configuration Description Language*

The first language is the homogeneous definition of the application architecture. The UML-based formalism introduced in the previous section was very close to the UML class diagram (rather than the UML component diagram, which can be quite confusing), reusing the concepts of classes, attributes and associations with multiplicities. However, the objective was mainly to reuse the expressiveness of the graphical notation, but with a domain specific semantic.
The DSML we introduce here proposes a simple intentional architecture description language which allows to reify the heterogenous structural architecture of the legacy level. We call this language the **Configuration Description Language** (CDL). The main subset of the metamodel is depicted in Figure 64.

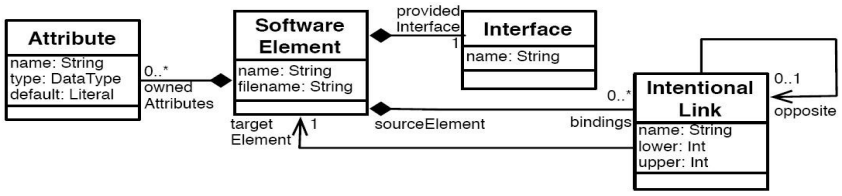


Figure 64: CDL metamodel

The main concept of this view is the **SoftwareElement** describing a particular type of software with its own configuration, management, and life cycle procedures. Each SoftwareElement is described by a set of properties (**ownedAttributes**), with an initial value (**defaultValue**), which are used by the administrator to reify the configurable attributes of the legacy software that the SoftwareElement represents. Note that a particular software can be reified by different SoftwareElements with different configuration properties. The configuration language allows to describe an architecture in intension. This means here that each described SoftwareElement can be deployed into several instances.
The architecture of the legacy level is intentionally reified through the definition of bindings (*IntentionalLink*), allowing to connect a SoftwareElement to another, and expressing a multiplicity (**lower** & **upper**) and a role (**name**). The multiplicity expresses the range of instances of the target SoftwareElement for each one of the source SoftwareElement. The role allows navigation with a query language relying on OCL. It is also possible to define bi-directional bindings by defining an opposite bindings.

e) *The Wrapping Description Language*

The second language allows the definition of a wrapper and its relation with SoftwareElements. In the Wrapping Description Language introduced in the previous section, a wrapper was described in an XML dialect, only enabling runtime checks by the Tune machinery. With this DSML, it becomes possible to introduce static consistency checks regarding the architecture schema it may reference, especially for the navigation clauses included in method parameters. The corresponding metamodel is presented in Figure 65, also with an illustration with a model defined with a specialized textual editor. A Wrapper describes **methods** which define actions that can be applied on the encapsulated software component. A wrapper may be referenced by different SoftwareElements (with different properties). A method can be parametrized (**ownedParameter**) with any property (of the SoftwareElement) of the configuration description in which the wrapper is used, the OCL-based navigation language allowing to fetch the effective parameter values. The method implementations (**imp**) are given in the form of a reference to a program (currently a string referring to a Java class).

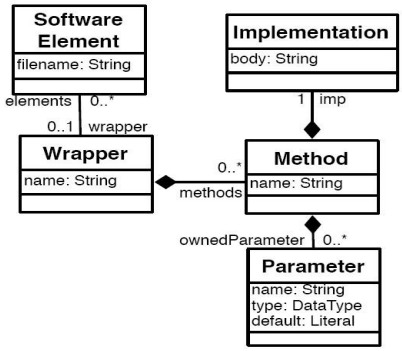


Figure 65: WDL metamodel

Note that this view must be consistent with the architectural view described with the CDL. We have thus defined OCL constraints to verify that the wrapper associated with a software element defines at least the methods provided by the interface.

f) The Deployment Description Language

The third language is used to define by intention or by extension, the real deployment of instances of each software component on system's nodes. In the UML-based formalisms described in the previous section, deployment policies were specified thanks to the *initial* and *host-family* attributes in the Architecture schema. The introduced deployment DSML is described by the metamodel presented in Figure 66. For this, we define for each SoftwareElement a set of **Deployments**, describing a real number of instances (**initial**) to be deployed on a node (**AbtractNode**). Nodes are known as "abstract" because they define a deployment policy (**policy**). Abstract nodes include the deployment information required to implement a deployment strategy, e.g. the physical address of a (single) real node on which instances should be deployed, or a list of physical addresses and an allocation function (for a cluster).

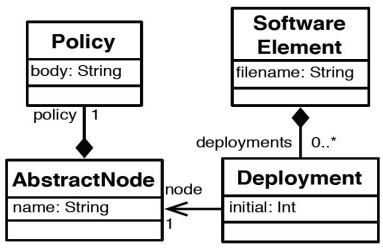


Figure 66: DDL metamodel

Note that this view must be consistent with the view described with the CDL. For instance, the number of deployed instances must be compatible with the multiplicities described in the configuration.
The clear separation of the deployment and architecture concerns allows to define several deployment orders for the same SoftwareElement (i.e. on different nodes), and to define different deployment models for the same architecture model. Finally, we clearly identify the concept of node.

g) The Reconfiguration Description Language

The last DSML allows the definition of reconfiguration policies. In the UML-based formalism introduced in the previous section, reconfiguration actions were expressed in terms of state diagrams, where we awkwardly inserted elementary actions as the state name. In accordance with the UML semantics, we decided to represent reconfiguration actions as activity diagrams in this DSML. Moreover, we introduced support in the DSML for the life cycle definition, allowing a clear description of links between events and reconfiguration actions, similarly to ECA (**Event/Condition/Action**) rules which typically used. This life cycle is expressed as a state diagram, where transitions are triggered by events and execute reconfiguration actions defined as activity diagram. We are inspired by the UML metamodel to express state diagrams (i.e. life cycles) and activity diagrams (i.e. reconfiguration policies). We don't reproduce this DSML metamodel since it is very similar to that of UML (with useless elements withdrawn).

9 Graphical Consoles

This chapter contains specifications related to the Task 1.8 of the SCOrWare project. This is dedicated to the design and implementation of SCOrWare graphical management consoles. Section 9.1 presents the SCOrWare heavy client approach for the SCOrWare Explorer graphical management console. The SCOrWare think client approach was not developed as the eXo Platform partner left the project.

9.1 SCOrWare Explorer

9.1.1 Objectives

FraSCAti Explorer is the SCA graphical management console of the SCOrWare project. It aims to provide a graphical management console allowing introspection of running SCA composites and dynamical reconfiguration. This tool is dedicated to the SCOrWare SCA runtime Tinfi, as other SCA runtimes don't take care of dynamical reconfiguration concerns.
Our objective with the FraSCAti Explorer is to provide an environment where SCA composites, SCA components, bindings can be easily discovered, monitored. Moreover, we will allow actions on the running infrastructure.
This objective is achieved by founding FraSCAti Explorer as a personality of the existing Fractal Explorer, the Fractal graphical management console. Fractal Explorer is itself a personality of the existing explorer tool, a framework to build generic graphical management consoles. We take advantage of the genuine feature of Fractal which allows dynamical reconfiguration of running components. FraSCAti Explorer user will have the ability to switch between SCA and Fractal views (roles).

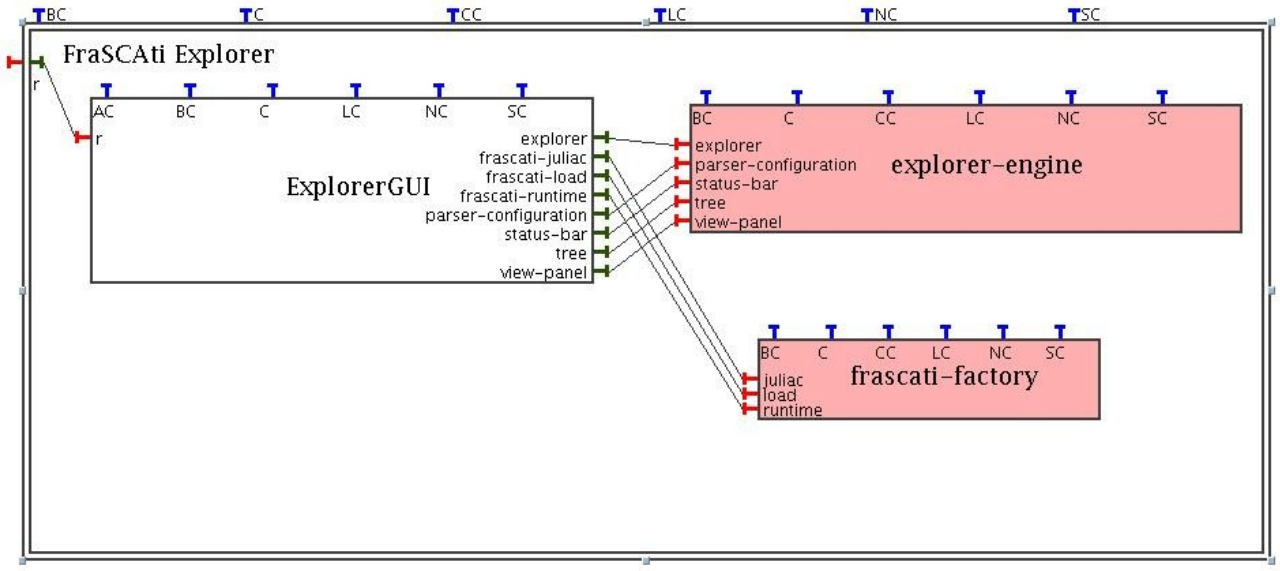
9.1.2 Specification

The FraSCAti Explorer framework is able to :

- load and start an SCA composite file (from a file system or inside a jar)
- explore a running SCA application
  - discover SCA composites
  - discover SCA components
  - discover SCA services and references
  - discover and monitor SCA properties
  - discover SCA bindings
  - discover SCA intents
- start/stop components or composites
- invoke methods
- switch between the SCA view and the Fractal View
- provide reconfiguration actions
  - add/remove component from composite
  - update component SCA properties
  - wire/unwire SCA components
  - update SCA bindings
  - add/remove SCA intents.

9.1.3 Architecture

The architecture is designed as a set of Fractal components. So, the FraSCAti Explorer itself can be displayed by the Fractal explorer. Here is a snapshot of the FraSCAti Explorer component.



You can see that the FraSCAti Explorer is a composite component hosting 3 components :

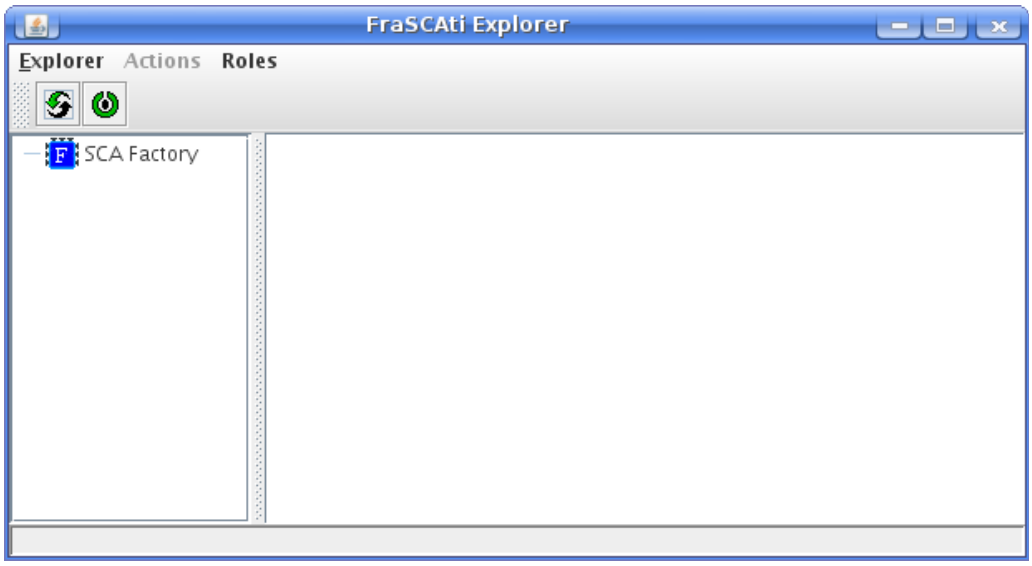
- the ExplorerGUI component in charge of application configuration and interactions between GUI and the SCA runtime,
- the explorer-engine component responsible for explorer configuration files parsing and GUI management (tree, panel, status-bar),
- the frascati-factory component which is itself the main component of the FraSCAti runtime allowing SCA composite description files loading, code injection, starting components, etc.

9.1.4 Implementation

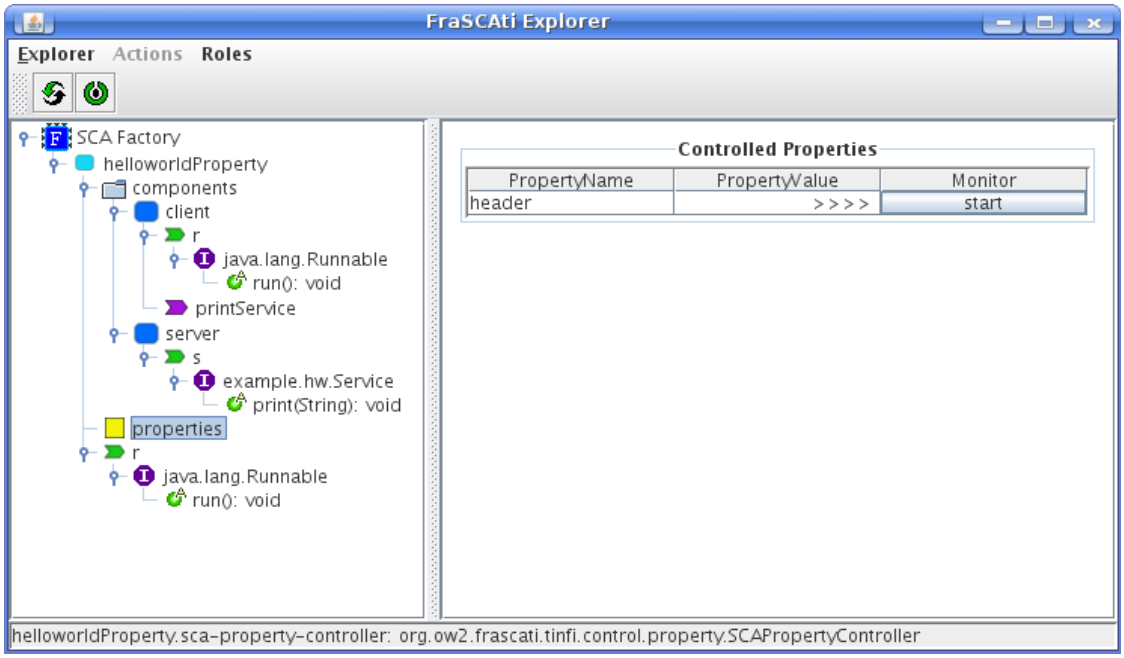
a) Design

The FraSCAti Explorer comes with the following standard layout :

- A menu bar allowing
  - to load explorer configuration files,
  - to exit the explorer,
  - to invoke actions on the selected tree entry
  - to switch between roles (SCA, Fractal)
- A tool bar (updated when a tree entry is selected)
- The left panel hosting the tree with the FraSCAti Assembly Factory as root component and loaded composites as children.
- The right panel used to display information depending of the selected tree entry.



Here is an example with a simple helloWorld example loaded in the FraSCAti Explorer.



b) *Plugins*

The FraSCAti Explorer can be configured with the help of a XML file describing for each node (type) :

- the associated icon
- associated contextual menu (actions available on the node) : display name and implementation class
- display panel : the panel (implementation class) used to display this node
- associated wrapper
- tooltips.

To illustrate the implementation of such a mechanism, we will describe how to define a new action on a node.

```
<node type-name="org.ow2.frascati.factory.core.dispatcher.Manager">
  <icon><icon-file url="icons/scaFactory.png"/></icon>
  <menu>
    <item label="Load">
      <code>org.ow2.frascati.explorer.action.LoadMenuItem</code>
    </item>
    <item label="Add to classpath ...">
      <code>org.ow2.frascati.explorer.action.AddToClasspathAction</code>
    </item>
  </menu>
  <wrapper>
    <code>org.ow2.frascati.explorer.context.AssemblyFactoryContext</code>
  </wrapper>
</node>
```

This example shows a configuration for the org.ow2.frascati.factory.core.dispatcher.Manager type. This type wrap the core component of the FraSCAti Explorer, the FraSCAti Factory. We can notice that an icon is defined for this type. We also define a contextual menu with two items “Load.” and “Add to classpath ...”.
Defining a new action in the explorer consists in two parts :

- add a menu item in the configuration file. It implies to fill in :
  - the menu label, here “Load ...”
  - the implementation class, here “org.ow2.frascati.explorer.action.LoadMenuItem”
- Write the implementation class.

The implementation class is a simple Java class that implements a generic interface MenuItem. This interface defines two methods :

- **int** getStatus (**final** TreeView treeView) : tells the GUI if the menu is enabled or disabled,
- **void** actionPerformed (**final** MenuItemTreeView e) : the action to execute when the action is invoked from the GUI.

The following figure is the definition of the “Load” menu item of the FraSCAti factory node.

```
public class LoadMenuItem
  implements MenuItem
{

 /** The {@link JFileChooser} instance. */
 protected static JFileChooser fileChooser = null;

 /*
  * @see org.objectweb.util.explorer.api.MenuItem#getStatus(org.objectweb.util.explorer.api.TreeView)
  */
 public int getStatus (final TreeView treeView) {
   return MenuItem.ENABLED_STATUS;
 }

 /*
  * @see org.objectweb.util.explorer.api.MenuItem#actionPerformed(org.objectweb.util.explorer.api.MenuItemTreeView)
```

```
 */
public void actionPerformed (final MenuItemTreeView e) {
 // At the first call, creates the file chooser.
 if(fileChooser == null) {
  fileChooser = new JFileChooser();
  // Sets to the user's current directory.
  fileChooser.setCurrentDirectory(new File(System.getProperty("user.dir")));
  // Filters SCA files.
  fileChooser.setFileFilter(
   new FileFilter() {
    /** Whether the given file is accepted by this filter. */
    public boolean accept(File f) {
     String extension = f.getName().substring(f.getName().lastIndexOf('.') + 1);
     return extension.equalsIgnoreCase("composite") || f.isDirectory();
    }
    /** The description of this filter. */
    public String getDescription() {
     return "SCA composite files";
    }
   }
  );
 }

 // Opens the file chooser to select a file.
 int returnVal = fileChooser.showOpenDialog(null);
 if (returnVal == JFileChooser.APPROVE_OPTION) {
  File f = fileChooser.getSelectedFile();
  String name = f.toString();
  int index = name.indexOf(".jar");

  if ( index != -1 ) { // File in a jar
   // Add the jar to the classpath
   URL[] urls = new URL[1];
   try {
    urls[0] = new File(name.substring(0, index+4)).toURI().toURL();
   } catch (MalformedURLException e1) {
    Log.warn(e1.getMessage());
   }
   ExplorerGUI.getSingleton().getFactoryLoadItf().loadJars(urls);
   // Load the composite
   loadComposite( name.substring(index + 5) );
  } else {
   loadComposite( f.getPath() );
  }
 }
}
```

```
...
}
```

We can notice that the menu item is always available. The action performed consists in opening a FileChooser box to let the user chooses a composite description file to load, and then to load it.

9.1.5 Summary

The FraSCAti Explorer aims to be the management tool of the FraSCAti runtime allowing to discover / monitor running SCA applications dynamically. Moreover, it provides a way to interact with running application by providing dynamical reconfiguration such as “hot component substitution” (unwire/remove/add/wire) , properties update, new binding generation, etc. This tool is not another tool built from scratch but takes advantage of frameworks used in various projects (explorer framework, fractal explorer) by building itself on top of such frameworks.
The FraSCAti Explorer will be useful to propose demonstrations, to test dynamical reconfiguration use cases, to validate SCA application development (architecture, ...). We can say that it is a first step to promote the FraSCAti runtime.

10 Conclusion

This document specifies the SCOrWare runtime platform, named FraSCAti, developed by the different SCOrWare partners. This includes:

- A state of the art related to service-oriented and component-based architectures, and more specifically SCA, JBI, and Fractal (Chapter 2),
- Corrected SCA schema and meta-model (Chapter 3),
- The Tinfi SCA runtime kernel (Section 4.1),
- The integration into JBI and PEtALS (Section 4.2),
- The Binding Factory (Chapter 5),
- The Semantic Trading Service (Chapter 7),
- The FraSCAti Assembly Factory (Section 8.1),
- SCA system deployment with FDF (Section 8.2), and
- Autonomic support with TUNe (Section 8.3).

FraSCAti is original related to other open source implementations, e.g., Apache Tuscany, as it is supported by both an EMF model-driven and a Fractal component-based approaches, and as it provides advanced features like fine grain reconfiguration of SCA composites, semantic trading, automatic deployment, and autonomous support.

The FraSCAti platform is hosted by the OW2 open source consortium and is available at http://frascati.ow2.org under LGPL license

11 References

[1] Andrei Popovici, Gustavo Alonso, Thomas R. Gross, *Spontaneous Container Services*, ECOOP, 2003.

[2] Apache Consortium, Tuscany, http://tuscany.apache.org/ .

[3] Areski Flissi, Philippe Merle, *A Generic Deployment Framework for Grid Computing and Distributed Applications*, Proceedings of the 2nd International OTM Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06), 2006.

[4] Cauldron, Newton, http://newton.codecauldron.org/.

[5] Codehaus, Fabric 3, http://xircles.codehaus.org/projects/fabric3.

[6] E. Bruneton, T. Coupaye, J.-B. Stefani, *The Fractal Component Model*, http://fractal.objectweb.org/specification, February 2004.

[7] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq,Vivien Quema, Jean-Bernard Stefani, *The Fractal Component Model and its Support in Java*, Software -- Practice and Experience (SP\&E), special issue on ``Experiences with Auto-adaptive and Reconfigurable Systems'', 1257-1284, 2006.

[8] FDF, Fractal Deployment Framework, http://fdf.gforge.inria.fr/.

[9] Frederic Duclos, Jacky Estublier, Philippe Morat, *Describing and using non functional aspects in component based applications*, AOSD, 2002.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming* , Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). LNCS 1241. pp. 220-242, June 1997.

[11] Java, Java Specification Request 235 : Service Data Objects.

[12] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien and Thierry Coupaye, *A Component Model Engineered with Components and Aspects*, In Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06), *2006.*

[13] Michael Clarke, Gordon S. Blair, Geoff Coulson, Nikos Parlavantzas, *An Efficient Component Model for the Construction of Adaptive Middleware*, Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, 2001.

[14] Open SOA, *Service Component Architecture - Assembly Model Specification*, 2007.

[15] Open SOA, *Service Component Architecture - Client and Implementation Model Specification for C++*, 2007.

[16] Open SOA, *Service Component Architecture - Client and Implementation Model Specification for WS-BPEL*, 2007.

[17] Open SOA, *Service Component Architecture - EJB Session Bean Binding Specification*, 2007.

[18] Open SOA, *Service Component Architecture - Java Common Annotations and API*, 2007.

[19] Open SOA, *Service Component Architecture - Java Component Implementation Specification*, 2007.

[20] Open SOA, *Service Component Architecture - JMS Binding Specification*, 2007.

[21] Open SOA, *Service Component Architecture - Spring Component Implementation Specification*, 2007.

[22] Open SOA, *Service Component Architecture - Web Service Binding Specification*, 2007.

[23] Roman Pichler and Klaus Ostermann and Mira Mezini, *On aspectualizing component models*, Softw., Pract. Exper., *2003*.

[24] Sacha Krakowiak, *Middleware Architecture with Patterns and Frameworks* : http://sardes.inrialpes.fr/~krakowia/MW-Book/.

[25] W3C, Document Object Model, http://www.w3.org/DOM/.

[26] W3C, Validator for XML Schema, http://www.w3.org/2001/03/webdata/xsv.

[27] W3C, Web service policy, http://www.w3.org/TR/ws{}-policy/.

[28] W3C, XML Schema, http://www.w3.org/XML/Schema.

[29] W3C, XSL Transformations, http://www.w3.org/TR/xslt.

[30] OW2 Consortium, ASM, http://asm.objectweb.org/.

[31] Eclipse Fondation, Eclipse Modeling Framework, http://www.eclipse.org/emf/.

[32] Sun Microsystems, Java Business Integration specification, http://www.jcp.org/en/jsr/detail?id=208.

[33] Sun Microsystems, Java Management, http://java.sun.com/products/JavaManagement.

[34] Open SOA Collaboration, http://www.osoa.org.

[35] EBM WebSourcing, PEtALS Getting Started, https://wiki.objectweb.org/petals/Wiki.jsp?page=GettingStarted.

12 Appendix

12.1 SCOrWare Model

12.1.1 sca-core.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  elementFormDefault="qualified">

  <element name="componentType" type="sca:ComponentType" />

  <complexType name="ComponentType">
    <sequence>
      <element ref="sca:implementation" minOccurs="0" maxOccurs="1" />
      <choice minOccurs="0" maxOccurs="unbounded">
        <element name="service" type="sca:ComponentService" />
        <element name="reference" type="sca:ComponentReference" />
        <element name="property" type="sca:Property" />
      </choice>
      <any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </sequence>
    <attribute name="constrainingType" type="QName" use="optional" />
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

  <element name="composite" type="sca:Composite" />

  <complexType name="Composite">
    <sequence>
      <choice minOccurs="0" maxOccurs="unbounded">
        <element name="include" type="sca:Include"/>
        <element name="service" type="sca:Service" />
        <element name="reference" type="sca:Reference" />
        <element name="component" type="sca:Component" />
        <element name="property" type="sca:Property" />
        <element name="wire" type="sca:Wire" />
      </choice>
      <any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="NCName" use="required" />
    <attribute name="targetNamespace" type="anyURI" use="optional" />
    <attribute name="local" type="boolean" use="optional" default="false" />
    <attribute name="autowire" type="boolean" use="optional" default="false" />
    <attribute name="constrainingType" type="QName" use="optional" />
    <attribute name="requires" type="sca:listOfQNames" use="optional" />
    <attribute name="policySets" type="sca:listOfQNames" use="optional" />
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

  <complexType name="Service">
    <complexContent>
      <extension base="sca:BaseService">
        <attribute name="promote" type="anyURI" use="required" />
      </extension>
    </complexContent>
  </complexType>

  <complexType name="BaseService">
    <sequence>
      <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
```

```
      <element name="operation" type="sca:Operation" minOccurs="0"
        maxOccurs="unbounded" />
      <element ref="sca:binding" minOccurs="0" maxOccurs="unbounded" />
      <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
      <any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="NCName" use="required" />
    <attribute name="requires" type="sca:listOfQNames" use="optional" />
    <attribute name="policySets" type="sca:listOfQNames" use="optional" />
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

  <element name="interface" type="sca:Interface" abstract="true" />

  <complexType name="Interface" abstract="true" />

  <complexType name="Reference">
    <complexContent>
      <extension base="sca:BaseReference">
        <attribute name="promote" type="sca:listOfAnyURIs" use="required" />
      </extension>
    </complexContent>
  </complexType>

  <complexType name="BaseReference">
    <sequence>
      <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
      <!-- <element name="operation" type="sca:Operation" minOccurs="0"
        maxOccurs="unbounded" /> -->
      <element ref="sca:binding" minOccurs="0" maxOccurs="unbounded" />
      <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
      <any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="NCName" use="required" />
    <attribute name="target" type="sca:listOfAnyURIs" use="optional" />
    <attribute name="wiredByImpl" type="boolean" use="optional" default="false" />
    <attribute name="multiplicity" type="sca:Multiplicity" use="optional"
      default="1..1" />
    <attribute name="requires" type="sca:listOfQNames" use="optional" />
    <attribute name="policySets" type="sca:listOfQNames" use="optional" />
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

  <complexType name="SCAPropertyBase" mixed="true">
    <!-- mixed="true" to handle simple type -->
    <sequence>
      <any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="1" />
      <!-- NOT an extension point; This xsd:any exists to accept
        the element-based or complex type property
        i.e. no element-based extension point under "sca:property" -->
    </sequence>
  </complexType>

  <!-- complex type for sca:property declaration -->
  <complexType name="Property" mixed="true">
    <complexContent>
      <extension base="sca:SCAPropertyBase">
        <!-- extension defines the place to hold default value -->
        <attribute name="name" type="NCName" use="required" />
        <attribute name="type" type="QName" use="optional" />
        <attribute name="element" type="QName" use="optional"/>
        <attribute name="many" type="boolean" default="false" use="optional" />
        <attribute name="mustSupply" type="boolean" default="false"
          use="optional" />
        <anyAttribute namespace="##other" processContents="lax" />
        <!-- an extension point ; attribute-based only -->
      </extension>
    </complexContent>
  </complexType>

  <complexType name="PropertyValue" mixed="true">
    <complexContent>
```

```
      <extension base="sca:SCAPropertyBase">
        <attribute name="name" type="NCName" use="required" />
        <attribute name="type" type="QName" use="optional" />
        <attribute name="element" type="QName" use="optional" />
        <attribute name="many" type="boolean" default="false" use="optional" />
        <attribute name="source" type="string" use="optional" />
        <attribute name="file" type="anyURI" use="optional" />
        <anyAttribute namespace="##other" processContents="lax" />
        <!-- an extension point ; attribute-based only -->
      </extension>
    </complexContent>
  </complexType>

  <element name="binding" type="sca:Binding" abstract="true" />

  <complexType name="Binding" abstract="true">
    <sequence>
      <!-- <element name="operation" type="sca:Operation" minOccurs="0"
        maxOccurs="unbounded" /> -->
    </sequence>
    <attribute name="uri" type="anyURI" use="optional" />
    <attribute name="name" type="NCName" use="optional" />
    <attribute name="requires" type="sca:listOfQNames" use="optional" />
    <attribute name="policySets" type="sca:listOfQNames" use="optional" />
  </complexType>

  <element name="bindingType" type="sca:BindingType" />

  <complexType name="BindingType">
    <sequence minOccurs="0" maxOccurs="unbounded">
      <any namespace="##other" processContents="lax" />
    </sequence>
    <attribute name="type" type="QName" use="required" />
    <attribute name="alwaysProvides" type="sca:listOfQNames" use="optional" />
    <attribute name="mayProvide" type="sca:listOfQNames" use="optional" />
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

  <element name="callback" type="sca:Callback" />

  <complexType name="Callback">
    <choice minOccurs="0" maxOccurs="unbounded">
      <element ref="sca:binding" />
      <any namespace="##other" processContents="lax" />
    </choice>
    <attribute name="requires" type="sca:listOfQNames" use="optional" />
    <attribute name="policySets" type="sca:listOfQNames" use="optional" />
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

  <complexType name="Component">
    <sequence>
      <element ref="sca:implementation" minOccurs="0" maxOccurs="1" />
      <choice minOccurs="0" maxOccurs="unbounded">
        <element name="service" type="sca:ComponentService" />
        <element name="reference" type="sca:ComponentReference" />
        <element name="property" type="sca:PropertyValue" />
      </choice>
      <any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="NCName" use="required" />
    <attribute name="autowire" type="boolean" use="optional" default="false" />
    <attribute name="constrainingType" type="QName" use="optional" />
    <attribute name="requires" type="sca:listOfQNames" use="optional" />
    <attribute name="policySets" type="sca:listOfQNames" use="optional" />
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

  <complexType name="ComponentService">
    <complexContent>
      <extension base="sca:BaseService" />
    </complexContent>
  </complexType>
```

```
<complexType name="ComponentReference">
  <complexContent>
    <extension base="sca:BaseReference">
      <attribute name="autowire" type="boolean" use="optional"
        default="false" />
    </extension>
  </complexContent>
</complexType>

<element name="implementation" type="sca:Implementation" abstract="true" />

<complexType name="Implementation" abstract="true">
  <attribute name="requires" type="sca:listOfQNames" use="optional" />
  <attribute name="policySets" type="sca:listOfQNames" use="optional" />
</complexType>

<element name="implementationType" type="sca:ImplementationType" />

<complexType name="ImplementationType">
  <sequence minOccurs="0" maxOccurs="unbounded">
    <any namespace="##other" processContents="lax" />
  </sequence>
  <attribute name="type" type="QName" use="required" />
  <attribute name="alwaysProvides" type="sca:listOfQNames" use="optional" />
  <attribute name="mayProvide" type="sca:listOfQNames" use="optional" />
  <anyAttribute namespace="##other" processContents="lax" />
</complexType>

<complexType name="Wire">
  <sequence>
    <any namespace="##other" processContents="lax" minOccurs="0"
      maxOccurs="unbounded" />
  </sequence>
  <attribute name="source" type="anyURI" use="required" />
  <attribute name="target" type="anyURI" use="required" />
  <anyAttribute namespace="##other" processContents="lax" />
</complexType>

<element name="include" type="sca:Include" />

<complexType name="Include">
  <attribute name="name" type="QName" use="required" />
  <anyAttribute namespace="##other" processContents="lax" />
</complexType>

<complexType name="Operation">
  <attribute name="name" type="NCName" use="required" />
  <attribute name="requires" type="sca:listOfQNames" use="optional" />
  <attribute name="policySets" type="sca:listOfQNames" use="optional" />
  <anyAttribute namespace="##other" processContents="lax" />
</complexType>

<element name="constrainingType" type="sca:ConstrainingType" />

<complexType name="ConstrainingType">
  <sequence>
    <choice minOccurs="0" maxOccurs="unbounded">
      <element name="service" type="sca:ComponentService" />
      <element name="reference" type="sca:ComponentReference" />
      <element name="property" type="sca:Property" />
    </choice>
    <any namespace="##other" processContents="lax" minOccurs="0"
      maxOccurs="unbounded" />
  </sequence>
  <attribute name="name" type="NCName" use="required" />
  <attribute name="targetNamespace" type="anyURI" use="optional"/>
  <attribute name="requires" type="sca:listOfQNames" use="optional" />
  <anyAttribute namespace="##other" processContents="lax" />
</complexType>

<simpleType name="Multiplicity">
  <restriction base="string">
    <enumeration value="0..1" />
    <enumeration value="1..1" />
```

```
      <enumeration value="0..n" />
      <enumeration value="1..n" />
    </restriction>
  </simpleType>

  <simpleType name="OverrideOptions">
    <restriction base="string">
      <enumeration value="no" />
      <enumeration value="may" />
      <enumeration value="must" />
    </restriction>
  </simpleType>

  <!-- Global attribute definition for @requires to permit use of intents
    within WSDL documents -->
  <attribute name="requires" type="sca:listOfQNames" />

  <!-- Global attribute defintion for @endsConversation to mark operations
    as ending a conversation -->
  <attribute name="endsConversation" type="boolean" default="false" />

  <simpleType name="listOfQNames">
    <list itemType="QName" />
  </simpleType>
  <simpleType name="listOfAnyURIs">
    <list itemType="anyURI" />
  </simpleType>
</schema>
```

12.1.2 sca-implementation-java.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0" elementFormDefault="qualified"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="2.0">

  <include schemaLocation="sca-core.xsd" />

  <element name="implementation.java" type="sca:JavaImplementation"
    substitutionGroup="sca:implementation" />
  <complexType name="JavaImplementation">
    <complexContent>
      <extension base="sca:Implementation">
        <sequence>
          <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
        </sequence>
        <attribute name="class" type="NCName" use="required" />
        <anyAttribute namespace="##any" processContents="lax" />
      </extension>
    </complexContent>
  </complexType>
</schema>
```

12.1.3 sca-implementation-composite.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  elementFormDefault="qualified">

  <include schemaLocation="sca-core.xsd" />
  <element name="implementation.composite" type="sca:SCAImplementation"
    substitutionGroup="sca:implementation" />
  <complexType name="SCAImplementation">
    <complexContent>
      <extension base="sca:Implementation">
        <sequence>
          <any namespace="##other" processContents="lax" minOccurs="0"
```

```
            maxOccurs="unbounded" />
        </sequence>
        <attribute name="name" type="QName" use="required" />
        <anyAttribute namespace="##any" processContents="lax" />
      </extension>
    </complexContent>
  </complexType>
</schema>
```

12.1.4 sca-binding-sca.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  elementFormDefault="qualified">

  <include schemaLocation="sca-core.xsd" />

  <element name="binding.sca" type="sca:SCABinding"
    substitutionGroup="sca:binding" />
  <complexType name="SCABinding">
    <complexContent>
      <extension base="sca:Binding">
        <!--<sequence>
          <element name="operation" type="sca:Operation" minOccurs="0"
          maxOccurs="unbounded" />
          </sequence>  -->
        <anyAttribute namespace="##any" processContents="lax" />
      </extension>
    </complexContent>
  </complexType>
</schema>
```

12.1.5 sca-binding-webservice.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:wsdli="http://www.w3.org/2004/08/wsdl-instance"
  xmlns:wsa="http://www.w3.org/2005/08/addressing"
  elementFormDefault="qualified"
  xmlns:tns="http://www.w3.org/2005/08/addressing">
  <import namespace="http://www.w3.org/2004/08/wsdl-instance"
    schemaLocation="http://www.w3.org/2004/08/wsdl-instance" />
  <import namespace="http://www.w3.org/2005/08/addressing"
    schemaLocation="http://www.w3.org/2006/03/addressing/ws-addr.xsd" />
  <include schemaLocation="sca-core.xsd" />
  <element name="binding.ws" type="sca:WebServiceBinding"
    substitutionGroup="sca:binding" />
  <complexType name="WebServiceBinding">
    <complexContent>
      <extension base="sca:Binding">
        <sequence>
          <element ref="wsa:EndpointReference" minOccurs="0"
            maxOccurs="unbounded" />
          <!-- <any namespace="##any" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/> -->
        </sequence>
        <attribute name="wsdlElement" type="anyURI" use="optional" />
        <attribute ref="wsdli:wsdlLocation" use="optional" />
        <anyAttribute namespace="##any" processContents="lax" />
      </extension>
    </complexContent>
  </complexType>
</schema>
```

12.1.6 sca-binding-jbi.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:wsdli="http://www.w3.org/2004/08/wsdl-instance"
  xmlns:wsa="http://www.w3.org/2005/08/addressing" elementFormDefault="qualified"
  xmlns:tns="http://www.w3.org/2005/08/addressing">
  <import namespace="http://www.w3.org/2004/08/wsdl-instance"
    schemaLocation="http://www.w3.org/2004/08/wsdl-instance" />
  <import namespace="http://www.w3.org/2005/08/addressing"
    schemaLocation="http://www.w3.org/2006/03/addressing/ws-addr.xsd" />
  <include schemaLocation="sca-core.xsd" />
  <element name="binding.jbi" type="sca:JBIBinding" substitutionGroup="sca:binding" />
  <complexType name="JBIBinding">
    <complexContent>
      <extension base="sca:Binding">
        <sequence>
          <choice>
            <element ref="wsa:EndpointReference" minOccurs="0" maxOccurs="unbounded" />
            <element ref="sca:ServiceReference" minOccurs="0" maxOccurs="unbounded" />
            <element ref="sca:InterfaceReference" minOccurs="0" maxOccurs="unbounded" />
          </choice>
        </sequence>
        <attribute ref="wsdli:wsdlLocation" use="optional" />
        <attribute name="wsdlElement" type="anyURI" use="optional" />
        <anyAttribute namespace="##any" processContents="lax" />
      </extension>
    </complexContent>
  </complexType>
  <element name="ServiceReference" type="string" />
  <element name="InterfaceReference" type="QName" />
</schema>
```

12.1.7 sca-policy.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  elementFormDefault="qualified">

  <include schemaLocation="sca-core.xsd" />
  <import namespace="http://schemas.xmlsoap.org/ws/2004/09/policy"
    schemaLocation="http://schemas.xmlsoap.org/ws/2004/09/policy/ws-policy.xsd" />

  <element name="intent" type="sca:Intent" />
  <complexType name="Intent">
    <sequence>
      <element name="description" type="string" minOccurs="0" maxOccurs="1" />
      <any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="QName" use="required" />
    <attribute name="constrains" type="sca:listOfQNames" use="required" />
    <attribute name="requires" type="sca:listOfQNames" use="optional" />
    <anyAttribute namespace="##any" processContents="lax" />
  </complexType>

  <element name="policySet" type="sca:PolicySet" />
  <complexType name="PolicySet">
    <choice minOccurs="0" maxOccurs="unbounded">
      <element name="policySetReference" type="sca:PolicySetReference" />
      <element name="intentMap" type="sca:IntentMap" />
      <element ref="wsp:PolicyAttachment" />
      <element ref="wsp:Policy" />
      <element ref="wsp:PolicyReference" />
      <!--  <any namespace="##other" processContents="lax"/> -->
    </choice>
    <attribute name="name" type="QName" use="required" />
    <attribute name="provides" type="sca:listOfQNames" use="optional" />
```

```
    <attribute name="appliesTo" type="string" use="required" />
    <anyAttribute namespace="##any" processContents="lax" />
  </complexType>

  <complexType name="PolicySetReference">
    <attribute name="name" type="QName" use="required" />
    <anyAttribute namespace="##any" processContents="lax" />
  </complexType>

  <complexType name="IntentMap">
    <choice minOccurs="1" maxOccurs="unbounded">
      <element name="qualifier" type="sca:Qualifier" />
      <any namespace="##other" processContents="lax" />
    </choice>
    <attribute name="provides" type="QName" use="required" />
    <attribute name="default" type="string" use="optional" />
    <anyAttribute namespace="##any" processContents="lax" />
  </complexType>

  <complexType name="Qualifier">
    <choice minOccurs="1" maxOccurs="unbounded">
      <element name="intentMap" type="sca:IntentMap" />
      <element ref="wsp:PolicyAttachment" />
      <!-- <any namespace="##other" processContents="lax"/> -->
    </choice>
    <attribute name="name" type="string" use="required" />
    <anyAttribute namespace="##any" processContents="lax" />
  </complexType>

  <element name="allow" type="sca:Allow" />
  <complexType name="Allow">
    <attribute name="roles" type="string" use="required" />
  </complexType>

  <element name="permitAll" type="sca:PermitAll" />
  <complexType name="PermitAll" />

  <element name="denyAll" type="sca:DenyAll" />
  <complexType name="DenyAll" />

  <element name="runAs" type="sca:RunAs" />
  <complexType name="RunAs">
    <attribute name="role" type="string" use="required" />
  </complexType>
</schema>
```

12.1.8 sca-definitions.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  elementFormDefault="qualified">
  <include schemaLocation="sca-core.xsd" />
  <include schemaLocation="sca-policy.xsd" />

  <element name="definitions">
    <complexType>
      <choice minOccurs="0" maxOccurs="unbounded">
        <element ref="sca:intent" />
        <element ref="sca:policySet" />
        <element ref="sca:binding" />
        <element ref="sca:bindingType" />
        <element ref="sca:implementationType" />
        <any namespace="##other" processContents="lax" minOccurs="0"
          maxOccurs="unbounded" />
      </choice>
      <attribute name="targetNamespace" type="anyURI" use="required" />
    </complexType>
  </element>

</schema>
```

12.2 Binding Factory Interfaces

12.2.1 Binding Factory component interface

```
public interface BindingFactory {
  /**
   * Export an interface using the specified hints.
   *
   * @param c
   *            the Fractal component owner of the interface to export
   * @param itfName
   *            the name of the server interface to export
   * @param hints
   *            hints used to export the interface
   *
   * @throws BindingFactoryException
   *             if some error occurs while exporting the interface
   */
  void export(Component c, String itfName, Map<String, Object> hints)
      throws BindingFactoryException;

  /**
   * Unexport an interface.
   *
   * @param c
   *            the Fractal component owner of the interface to unexported
   * @param itfName
   *            the name of the server interface to unexport
   * @param hints
   *            hints to be used to unexport the interface
   * @throws BindingFactoryException
   *             if some error occurs while unexporting the interface
   */
  void unexport(Component c, String itfName, Map<String, Object> hints)
      throws BindingFactoryException;

  /**
   * Bind the specified client interface using the specified hints.
   *
   * @param component
   *            the Fractal component owner of the interface to bind
   * @param itfName
   *            the name of client interface to bind
   * @param hints
   *            a map of hints to reconstruct the export id
   * @throws BindingFactoryException
   *             if some error occurs while binding the interface
   */
  void bind(Component component, String itfName, Map<String, Object> hints)
      throws BindingFactoryException;

  /**
   * Unbind the specified client interface.
   *
   * @param component
   *            the Fractal component owner of the interface to unbind
   * @param clientItf
   *            the name of the client interface to unbind.
   * @throws BindingFactoryException
   *             if some error occurs while unbinding.
   */
  void unbind(Component component, String clientItf)
      throws BindingFactoryException;
}
```

12.2.2 Binding Factory component interface

```
/**
 * <p>
 * An interface to model a binding factory plugin. It exposes low-level actions
 * used by the default implementation of the {@link BindingFactory} to create
 * stub and skeleton components.
 * </p>
 * During the export phase (see
 * {@link BindingFactory#export(Component, String, Map)}, a skeleton is created,
 * then it is finalized, and then it is started.<br>
 * The same actions are performed for stub components.
 *
 * @see BindingFactory
 * @See {@link BindingFactoryImpl} the default implementation
 */
public interface BindingFactoryPlugin<E extends ExportHints, B extends BindHints> {

  public static final String IDENTIFIER = "identifier";

  /**
   * Creates plugin-specific skeletons.
   *
   * @param interfaceName
   *            the name of the interface to export
   * @param interfaceToExport
   *            a reference to the the interface to export
   * @param owner
   *            the component owning the interface to export
   * @param exportHints
   *            hints used to create the skeleton
   * @return the skeleton component
   * @throws SkeletonGenerationException
   *             if some error occurs while generating the skeleton
   */
  Component createSkel(String interfaceName, Object interfaceToExport,
      Component owner, E exportHints) throws SkeletonGenerationException;

  /**
   * Creates plugin specific stubs.
   *
   * @param client
   *            the component owning the interface to export
   * @param itfname
   *            the name of the client interface to bound to the stub
   * @param hints
   *            hints to create the stub
   * @throws StubGenerationException
   *             if some error occurs while generating the stub
   */
  Component createStub(Component client, String itfname, BindHints hints)
      throws StubGenerationException;

  /**
   * Finalize the stub component, setting bindings or performing operations
   * that couldn't be done before.
   *
   * @param stub
   *            the stub component to finalize
   * @param hints
   *            hints to be used during the finalization process
   * @throws BindingFactoryException
   */
  void finalizeStub(Component stub, Map<String, Object> hints)
```

```
        throws BindingFactoryException;

    /**
     * Finalize the skel component, setting bindings or performing operations
     * that couldn't be done before.
     *
     * @param skel
     *            the skeleton component to finalize
     * @param finalizationHints
     *            hints to be used during the finalization process
     * @throws BindingFactoryException
     */
    void finalizeSkeleton(Component skel, Map<String, String> finalizationHints)
        throws BindingFactoryException;

    /**
     * The stub component factory required by this plug in.
     *
     * @return stub component factory required by this plug in.
     */
    StubComponentFactory getStubComponentFactory();

    /**
     * @param initialHints
     *            the hints passed to the binding factory
     * @return the {@link ExportHints} specific for this connector.
     */
    E getExportHints(Map<String, Object> initialHints);

    /**
     *
     * @param initialHints
     *            the hints passed to the binding factory
     *
     * @return the {@link BindHints} specific for this connector.
     */
    BindHints getBindHints(Map<String, Object> initialHints);

    /**
     * The unique identifier for this plugin.
     *
     * @return a string identifier for this plugin
     */
    String getPluginIdentifier();

}
```

12.3 FDF Example

```
# Description of hosts with Java
Hosts = INTERNET.NETWORK {

  m1 = INTERNET.HOST {
    hostname = INTERNET.HOSTNAME(m1);
    user     = INTERNET.USER(admin,password,key);
    transfer = TRANSFER.SCP;
    protocol = PROTOCOL.OpenSSH;
    shell    = SHELL.SH;
    software {
        java = JAVA.JRE {
          archive = JAVA.ARCHIVE(/archives/jre.zip);
          home    = JAVA.HOME(/tmp/jre);
        }
      }
    }
  }

  m2 = INTERNET.HOST {
    hostname = INTERNET.HOSTNAME(m2);
    user     = INTERNET.USER(admin,password,key);
    transfer = TRANSFER.SCP;
    protocol = PROTOCOL.OpenSSH;
    shell    = SHELL.SH;
    software {
        java = JAVA.JRE {
          archive = JAVA.ARCHIVE(/archives/jre.zip);
          home    = JAVA.HOME(/tmp/jre);
        }
      }
    }
  }

  m3 = INTERNET.HOST {
    hostname = INTERNET.HOSTNAME(m3);
    user     = INTERNET.USER(admin,password,key);
    transfer = TRANSFER.SCP;
    protocol = PROTOCOL.OpenSSH;
    shell    = SHELL.SH;
    software {
        java = JAVA.JRE {
          archive = JAVA.ARCHIVE(/archives/jre.zip);
          home    = JAVA.HOME(/tmp/jre);
        }
      }
    }
  }

  m4 = INTERNET.HOST {
    hostname = INTERNET.HOSTNAME(m4);
    user     = INTERNET.USER(admin,password,key);
    transfer = TRANSFER.SCP;
    protocol = PROTOCOL.OpenSSH;
    shell    = SHELL.SH;
    software {
        java = JAVA.JRE {
          archive = JAVA.ARCHIVE(/archives/jre.zip);
          home    = JAVA.HOME(/tmp/jre);
        }
      }
    }
  }
}

# Description of PEtALS on M1

petals-on-M1 = PEtALS.SERVER {
  archive = PEtALS.ARCHIVE(/archives/petals.zip);
  home    = PEtALS.HOME(/tmp/petals);
```

```
  host    = Hosts/m1;
}

frascati-jbi = FRASCATI.SERVERonPEtALS {
  archive = PEtALS.ARCHIVE(/archives/frascati-jbi.zip);
  petals  = /petals-on-M1;
}

# Description of SCA application C1 on  PEtALS server

C1 = FRASCATI.COMPOSITEonPEtALS {
  archive = PEtALS.ARCHIVE(/archives/c1.zip);

  configuration {
    reference = SCA.REFERENCE(R1) {
     binding = C2/configuration/service;
    }
    P1 = SCA.PROPERTY(p1,value);
  }
}

# Description of FraSCAti server on M2

frascati = FRASCATI.SERVER {
  archive = FRASCATI.ARCHIVE(/archives/frascati.zip);
  home = FRASCATI.HOME(/tmp/frascati);
  host = Hosts/m2;
}

# Description of SCA application C2 on  FraSCAti server

C2 = FRASCATI.COMPOSITE(c2.zip) {
  runtime = /frascati;

  configuration {
    service = SCA.SERVICE(S2) {
      binding = SCA.BINDING.RMI(m2, 1099, c2s2);
    }
    reference = SCA.REFERENCE(R2) {
      binding = C3/configuration/binding;
    }
    P2 = SCA.PROPERTY(p2,value);
    P3 = SCA.PROPERTY(p3,value);
  }
}

# Description of Tuscany standalone on M3

tuscany = TUSCANY.RUNTIME {
  archive = TUSCANY.ARCHIVE(/archives/tuscany.zip);
  home = TUSCANY.HOME(/tmp/tuscany);
  host = Hosts/m3;
}

# Description of SCA application C3 on Tuscany

C3 = TUSCANY.COMPOSITE(Main, c3.zip) {
  runtime = /tuscany;

  configuration {
    service = SCA.SERVICE(S3) {
     binding = SCA.BINDING.RMI(m3, 1099, c3s3);
    }
    reference = SCA.REFERENCE(R3) {
      binding = C4/configuration/binding;
    }
  }
}

# Description of Tomcat server on M4

tomcat-on-M4 = TOMCAT.SERVER {
  archive = TOMCAT.ARCHIVE(/archives/tomcat.zip);
  home    = TOMCAT.HOME(/tmp/tomcat);
  host    = Hosts/m4;
}
```

```
# Description of SCA application C4 on Tomcat

C4 = TUSCANY.WAR {
  archive = TOMCAT.WAR.ARCHIVE(c4.war);
  name = TOMCAT.WAR.NAME(c4);
  tomcat  = tomcat-on-M4;

  configuration {
    service = SCA.SERVICE(S4) {
     binding = SCA.BINDING.RMI(m4, 1099, c4s4);
    }
    P4 = SCA.PROPERTY(p4,value);
  }
}
```