SCA Service Component Architecture

Java EE Integration Specification

SCA Version 1.00, May 13 2008

Technical Contacts:

Ron Barack	SAP AG
Michael Beisiegel	IBM Corporation
Henning Blohm	SAP AG
Dave Booz	IBM Corporation
Mike Edwards	IBM Corporation
Anish Karmarkar	Oracle Corporation
Michael Keith	Oracle Corporation
Ashok Malhotra	Oracle Corporation
Sanjay Patil	SAP AG
Prasad Peddada	BEA Systems, Inc.
Peter Peshev	SAP AG
Matthew Peters	IBM Corporation
Michael Rowley	BEA Systems, Inc.

Copyright Notice

License

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

Table of Contents

1 Introduction

This document specifies the use of Service Component Architecture (SCA) within and over the scope of applications and modules developed, assembled, and packaged according to the Java Platform Enterprise Edition (Java EE) specification.

Java EE is the standard for Java-based enterprise applications today. While it offers a rich set of technologies, it does not define important concepts that are inherently required in service oriented architectures such as

- Extensibility of component implementation technologies
- Extensibility of transport and protocol abstractions
- a notion of cross-application assembly and configuration

The Service Component Architecture on the other hand provides a standardized and extensible assembly language and methodology that can be layered on top of existing component models and runtimes.

While the Java EE client and implementation specification will focus on the projection of SCA's concepts of assembly, implementation type, and deployment onto Java EE structures, it is expected that SCA application assemblies will combine Java EE components with other technologies. Examples of technologies for which SCA integration specifications have been completed include BPEL and the Spring framework. It is expected that an ***SCA enabled Java EE runtime*** will offer a palette of technologies for integration in an SCA assembly.

This specification defines the integration of SCA and Java EE within the context of a Java EE application, the use of Java EE components as service component implementations, and the deployment of Java EE archives either within or as SCA contributions. It is also possible to use bindings to achieve some level of integration between SCA and Java EE. These bindings are addressed in separate specifications: The EJB Session Bean Binding Specification [2] describes the exposure and consumption session beans; the JMS Binding Specification [9] describes the exposure and consumption of Java Message System (JMS) destinations; and a Binding Specification for Java Connectivity Architecture (JCA) adaptors should be published in the near future (as of this writing).

2 Scenarios

As already informally introduced above, we will use the term ***SCA-enabled Java EE runtime*** to refer to a Java EE runtime that supports deployment and execution of SCA-enhanced Java EE applications as well as SCA-enhanced Java EE modules (see also section 6).

An SCA-enabled Java EE runtime that fully implements this specification would support the use cases defined in appendix A. They are demonstrating the following scenarios:

2.1 Consume SCA-exposed services from Java EE components

For example, a web component should be able to easily consume a service implemented by a service component, either by using SCA constructs in the implementation of a Java EE component implementation or via an EJB reference in combination with an EJB binding as defined in [2] over an SCA service.

2.2 Use Session Beans as Service Component Implementations

The recursive assembly model of SCA provides rich means of configuration and re-use of service components that may be implemented as SCA composites or by some other implementation type. Session beans are the Java EE component implementation model and serve also as service component implementations.

2.3 Expose Enterprise Applications into an SCA domain

The SCA Assembly specification describes a deployment model for SCA contributions that provides cross-enterprise application assembly capabilities when layered over Java EE.

2.4 Use Recursive SCA Assembly in Enterprise Applications

SCA Assembly provides means to define sophisticated application assembly for enterprise applications.

2.5 Deploy SCA Components as a Part of a Java EE application

SCA applications will typically combine Java EE components with components using other implementation technologies, such as BPEL. This specification enables the deployment of components implemented in these “foreign” technologies as part of a Java EE application, taking advantage of whatever tooling and infrastructure support exists for the deployment and lifecycle management of Java EE applications. Such components are treated as running in unmanaged environment and should not rely on Java EE features (access to java:comp/env, etc.)

2.6 Use Java EE Archives as Service Component Implementation

This specification enables the creation of SCA applications whose components are implemented by Java JEE archives, so that they can be wired to each other and to components implemented using other technologies. This use-case requires a high-level view of the Java EE application as a single SCA component implementation, providing services and consuming references as a single component.

3 Overview of SCA Assembly in a Java Enterprise Edition Environment

This specification defines a model of using SCA assembly in the context of a Java EE runtime that enables integration with Java EE technologies on a fine-grained component level as well as use of Java EE applications and modules in a coarse-grained large system approach.

The Java EE specifications define various programming models that result in application components, such as Enterprise Java Beans (EJB) and Web applications that are packaged in modules and that are assembled to enterprise applications using a Java Naming and Directory Interface (JNDI) based system of component level references and component naming.

Names of Java EE components are scoped to the application package (including single module application packages), while references, such as EJB references and resource references, are scoped to the component and bound in the Environment Naming Context (ENC).

In order to reflect and extend this model with SCA assembly, this specification introduces the concept of the Application Composite (see section 6.1.3) and a number of implementation types, such as the EJB implementation type and the Web implementation type, that represent the most common Java EE component types (see section 5).

Implementation types for Java EE components associate those component implementations with SCA service components and their configuration, consisting of SCA wiring and component properties as well as an assembly scope (i.e. a composite). Note that the use of these implementation types does not create new component instances as far as Java EE is concerned. Section 3.1 explains this in more detail.

In terms of packaging and deployment this specification supports the use of a Java EE application package as an SCA contribution, adding SCA's domain metaphor to regular Java EE packaging and deployment.

In addition, the JEE implementation type provides a means for larger scale assembly of contributions in which a Java EE application forms an integrated part of a larger assembly context and where it is viewed as an implementation artifact that may be deployed several times with different component configurations. See section 7 for more details.

Through the extended semantics of the application composite and by virtue of the component type definition for the JEE implementation type, both approaches, local assembly within the Java EE package as well as a coarse-grained use, can be combined without introducing model friction.

3.1 Life-Cycle Model for Service Components from Java EE Components

The EJB implementation type and the Web implementation type differ from other SCA implementation types in that they refer to components whose life cycle is not completely controlled by the SCA runtime implementation but rather in a shared responsibility with a Java EE runtime.

This model is motivated by several considerations:

- EJB and Web components may be invoked out-of-band from an SCA perspective: for example via a JNDI lookup and invocation in the case of a session bean, by receiving a JMS message in the case of a Message-Driven bean, or by an HTTP request in the case of a web application.

- Prior to invocation of an SCA enhanced component, the runtime must provide the Java EE context for the Java EE components as well as the SCA context (e.g. by injecting references)..

This specification defines the following rules that eliminate potential ambiguities:

- A Java EE component must not be used more than once as implementation of an SCA service component within the assembly of a Java EE application package (an EAR archive, or a standalone web application module, or a standalone EJB module).
- If a Java EE component that has a component type side file and/or is enhanced by SCA annotations is not used as a component implementation by an explicit service component declaration within the assembly of a Java EE application package, then it will not be associated with a component context and any SCA annotation may cause an error or may be ignored.

Furthermore the following life cycle handling rules apply:

- The component life cycle of an SCA enhanced Java EE component (see [4]) is nested within its Java EE component life cycle. More specifically:
  - Java EE initialization of an SCA enhanced Java EE component will happen before any SCA component initialization. Both occur before any business method invocation (or HTTP request in the case of a web application).
  - If an EJB has a PostConstruct interceptor registered, component initialization will happen before the interceptor is called.
  - No business method invocation (or HTTP request in the case of a web application) on the service component will occur after scope destruction (i.e. while and after @Destroy life cycle methods are called) and before the component implementation instance is finalized.
- The point in time of deployment of an SCA enhanced Java EE component is exactly the point in time it is deployed as a Java EE component.

3.2 Mapping a Java EE Component's Environment to Component Type Data

In the absence of optional extensions, the component type of a Java EE component (such as a Servlet or Enterprise Bean) does not contain SCA references. However, as an optional extension, an SCA runtime can choose to provide the capability of re-wiring EJB references using SCA. If an SCA runtime provides this optional extension, then the following rule is applied:

Each EJB 3 remote reference of each session bean within the Java EE application is exposed as an SCA reference. Each EJB reference has a target (within the Java EE application) that is the EJB identified by the configuration metadata within the JEE application - it is this target which may be overridden by a new target identified in the SCA metadata of the component using the JEE application.   The multiplicity of the generated reference is 0..1. The generated reference must require the “ejb” intent :

<intent name="ejb" constrains="sca:binding">

<description> The EJB intent requires that all of the semantics required by the Java EE specification for a communication to or from an EJB must be honored </description>

</intent>

As an additional vendor extension, each environment entry with a simple type may be translated into an SCA property. The name of the property is derived from the name of the resource, according to the algorithm given below. The XML simple type of the SCA property is derived from the Java type of the environment entry according to the following type mapping:

Environment Entry Type	XSD Type
String	String
Character	String
Byte	Byte
Short	Short
Integer	Int
Long	Long
Boolean	Boolean
Double	Double
Float	Float

Note that SCA names for references are of the XML Schema type NCName, while Java EE names for EJB references are of a type that allows a larger character set than what is supported in NCNames. The following escape algorithm defines how to translate names of EJB references and into names of SCA references:

1. Replace all “/” characters by “_” (underscore) characters
2. All remaining characters that are not supported in NCName are escaped as XML entities or character references.

These optional extensions are in no way required to be provided by any given SCA runtime and that, as a result, it is unadvisable to rely on the capability of rewiring EJB references when porting applications between different runtimes.

4 Scope and Limitations of the Specification

Various parts of this specification are limited with respect to what version of Java EE specifications they refer and apply to.

- <implementation.ejb/> is only defined for EJB version 3 and higher.
- <implementation.web/> is only defined for Servlet JSP specification version 2.5 and higher.
- <implementation.jee/> is only defined for Java EE archives that are compliant to Java EE 5 and higher

5 Java EE Component Based Implementation Types

The elementary building block of SCA assembly is the Service Component. In order to provide first class capabilities for exposure of services or consumption of service components, we define implementation types that represent the most prominent application component in Java EE applications: Enterprise JavaBeans (EJB) and Web application components.

The intention is to define a convenient implementation model for developers of these components. For example, a web component developer can use SCA annotations such as ***@Reference*** to declare service component references in the web component implementation.

5.1 Using Session Beans as Implementation Types

Session beans are the Java EE means to encapsulate business logic in an environment that manages remoting, security, and transaction boundaries. Service components play a similar role in SCA and so session beans are the most obvious candidates for service component implementation in a Java EE environment.

The SCA service programming model described in [5] resembles the EJB 3.0 programming model, for instance in its use of dependency injection. As in EJB 3.0, and unlike EJB 2.x, service interfaces do not need to extend any framework defined interfaces. An SCA-enabled Java EE runtime MUST support EJB 3.0 session beans as implementation types. An SCA-enabled Java EE runtime is not required to support EJB 2.1 session beans as SCA component implementation types. Handling of other JavaEE components, such as Message Driven Beans, is discussed in later sections.

Services and references of service components are associated with interfaces that define the set of operations offered by a service or required by a reference when connecting (“wiring”) with other services and references directly or via bindings. Interface definitions are hence an important part of the assembly meta-data and we need to define the particularities of interfaces derived from Java EE components

5.1.1 Mapping EJB business Interfaces to SCA Service Interfaces

The service interface derived from the business interface of an EJB 3 session bean is comprised of all methods of the EJB business interface. Furthermore:

> The service interface is remotable if and only if it is derived from a remote business interface. The EJB semantics for remote and local invocations (and thus the by-reference and by-value calls) as defined in [8] must be honored .

In the case of a business interface of a stateful session bean:

- The service interface is treated as conversational
- Methods of the interface that are implemented by @Remove methods are treated as @EndsConversation methods of the interface.

5.1.2 The Component Type of an Unaltered Session Bean

The component type of a session bean that does not use any SCA annotation and is not accompanied by a component type side file is constructed according to the following algorithm:

1. Each EJB 3 business interface of the session bean translates into a service by the unqualified name of the interface according to section 5.1.1. Such generated services require the EJB intent

(i.e. they are treated as if there was @requires=”ejb” definition in the business interface). EJB 2.x component interfaces are ignored.

2. Remote EJB 3 references MAY translate into an SCA references according to section 3.2.
3. Each Simple-Typed Environment Entry of the session MAY translate into an SCA property according to section 3.2.

For example:

```
package services.accountdata;

import javax.ejb.Local;

@Remote
public interface AccountService {
        AccountReport getAccountReport(String customerId);
}
```

with a session bean implementation

```
package services.accountdata;

import javax.ejb.Stateless;

@Stateless
public class AccountServiceImpl implements AccountService {

        public AccountReport getAccountReport(String customerId) {
                // ...
                return null;
        }
}
```

would result in the following component type:

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
  <service name="AccountService">
    <interface.java interface="services.accountdata.AccountService"/>
  </service>
</componentType>
```

5.1.3 Dependency Injection

Any session bean (or other Java EE construct) that is serving as the implementation type of an SCA service component may use dependency injection to acquire handles to the services wired to the component by the SCA assembly. Dependency injection may also be used to obtain the value of properties, a handle to the ComponentContext, a reference to the callback service and attributes of the current conversation. The following table shows the annotations that may be used to indicate the fields or properties to be injected.

Annotation	Purpose
@Callback	Session beans only: Mark method/field for callback injection
@ComponentName	Injection of component name
@Context	Injection of SCA context into member variable of service component instance
@Property	Injection of configuration properties from SC configuration
@Reference	Injection of Service references. There is no requirement that an SCA reference would appear under java:comp/env.
@ConversationID	Stateful Session beans only: Injection of a conversation id

A complete description of these annotations, and the values associated with them, is given in the Java Common Annotations and APIs specification [5].

When a session bean uses dependency injection, the container MUST inject these references after the bean instance is created, and before any business methods are invoked on the bean instance. If the bean has a PostConstruct interceptor registered, dependency injection MUST occur before the interceptor is called.

EJB's dependency injection occurs as part of construction, before the instance processes the first service request. For consistency, SCA's dependency injection also occurs during this phase. Instances of stateless session beans are typically pooled by the container. This has some consequences for the programming model for SCA.

In general, the values returned from the injected ComponentContext must reflect the current state in which the SCA component is being called. In particular, the value of getRequestContext() MUST return the request context of the current service call, not the request context for which the bean was initially created.

See also section 3.1 for an overview over the life cycle handling of SCA-enhanced Java EE components.

5.1.4 Providing additional Component Type data for a Session Bean

Several of the annotations described in [4] influence the implied component type of the session bean (or other Java EE construct). The following table shows the annotations that are relevant in a SCA-enabled Java EE runtime.

Annotation	Purpose
@Property	Adds a property to the implied component type. The type of the property is obtained through introspection.

@Reference	Adds a reference to the implied component type. The interface associated with this wire source is obtained through introspection. In the case a field is annotated with both @EJB and @Reference, SCA wiring overrides the EJB target identified by the configuration metadata within the JEE application by a new target according to SCA wiring rules. If the SCA reference is not wired, the value of the field is the target EJB as determined by Java EE semantics.
@Service	Session beans only: Allows the specification of which of the bean's EJB business interfaces should be exposed as SCA services. The business interface indicated in this annotation MUST BE EJB 3 compliant business interface. The service name of the implied component service will be the unqualified name of the interface. A remote interface is considered a remotable SCA interface. If the @Service annotation is not used, component services will be generated for each business interface exposed by the bean, as described in the section on the component type of unannotated Session Beans.

An SCA-enabled Java EE runtime MUST observe the specified annotations and use them when generating an implied component type.

Note that the set of annotations relevant to Java EE is a subset of those defined in [4]. Many of the remaining annotations duplicate functionality already available using Java EE annotations. An example is SCA's @Remotable tag, which duplicates functionality already available using Java EE's @Remote tag. To prevent redundancies and possible inconsistencies, the annotations given in [4] but not listed in the above table MUST be ignored.

5.1.4.1 Example of the use of annotations:

Using annotations, it is easy to create a component with a more complex component type. Continuing the example from section 3.1.1, we now add properties and references that can be injected based on the components use in an SCA assembly.

```
package services.accountdata;

import javax.ejb.Stateless;
import org.osoa.sca.annotations.*;

import services.backend.BackendService;

@Stateless
public class AccountServiceImpl implements AccountService {
        @Reference protected BackendService backend;
        @Property protected String currency;

        public AccountReport getAccountReport(String customerId) {
                // ...
                return backend(customerId, currency);
        }
}
```

would result in the following component type:

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
  <service name="AccountService">
    <interface.java interface="services.accountdata.AccountService"/>
  </service>
  <property name="currency"/>
  <reference name=”backend”>
    <interface.java interface="services.backend.BackendService"/>
  </reference>
</componentType>
```

5.1.5 Using a ComponentType Side-File

Using SCA annotations, a service component developer can easily create session beans that imply a complex component type. If further tuning of the component type is necessary, a component type side file may be included in the contribution. The component type side file follows the naming pattern

META-INF/<bean name>.componentType

and is located in the ejb module containing the bean. The rules on how a component type side file adds to the component type information reflected from the component implementation are described as part of the SCA assembly model specification [3]. If the component type information is in conflict with the implementation, it is an error as defined in [3].

If the component type side file specifies a service interface using a WSDL interface, then the bean interface MUST be compliant with the specified WSDL, according to the rules given in section 'WSDL 2 Java and Java 2 WSDL' in the Java Annotations and APIs Specification [4].

Use of the side file is recommended in cases where the ComponentContext API will be used instead of dependency injection to obtain service references. Since there is no annotation, introspection will not be able to see the need to insert a new reference into the component type.

5.1.6 Creating SCA components that use Session Beans as Implementation Types

In order to declare a service component instance that is implemented as a session bean, an ***implementation.ejb*** declaration can be put in some composite definition (see below). It has the following pseudo schema:

```
<implementation.ejb ejb-link="<ejb-link-name>”/>
```

The ejb-link-name attribute uniquely identifies the EJB that serves as the component implementation. The format of the value is identical to the format of the ***ejb-link*** tag in a Java EE deployment descriptor. In the case that the SCA contribution containing the composite file is an application EAR file, it is possible that several session beans have the same name. In that case the value of the ejb-link element must be composed of a path name specifying the ejb-jar containing the referenced enterprise bean with the ejb-name of the referenced enterprise bean appended and separated from the path name with a ‘#’. The path name is relative to the root of the EAR. In the case that SCA contribution is an EJB module's JAR file, the path name may generally be omitted.

The following example declares a service component named ***beancomponent*** in the composite ***beancomposite*** of the namespace ***http://www.sample.org***. ***Beancomponent*** is implemented by the bean ***SimpleBean*** in the ejb-module ***module.jar***. ***Beancomponent*** exposes a service, named after the bean's business interface name, that is promoted to the composite level:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="beancomposite" targetNamespace="http://www.sample.org"
        xmlns="http://www.osoa.org/xmlns/sca/1.0">

        <service name="AccountReporting" promote="beancomponent/AccountService"/>

        <component name="beancomponent">
                <implementation.ejb ejb-link="module.jar#SimpleBean"/>
        </component>
</composite>
```

5.1.7 Limitations on the use of Session Beans as Component Implementation

Session beans that serve as SCA implementations are none-the-less session beans, and may be found and used just like any other session bean, for instance, through dependency injection via an @EJB annotation, or though JNDI lookup.

An enterprise bean accessed through normal Java EE methods can contain SCA annotations such as @Reference or @Property, or may look up its configuration through the API, and therefore, require configuration from the SCA runtime.

Therefore, within the assembly of the contribution package, a session bean may be used as service component implementation at most once. Whether the enterprise bean is accessed through standard Java EE means, or through an SCA reference, the same service component configuration is used (see also section 3).

The EJB Specification defines a container contract that defines what behavior implementations may expect from the container, and what behavior the container can expect from the implementation. For instance, implementations are forbidden from managing class loaders and threads, but on the other hand, implementations need not be programmed for thread safety, since the container guarantees that no bean instance will be accessed concurrently. In an SCA-enabled Java EE runtime, both parties are expected to continue to abide by this contract. That is, a session bean that is serving as an SCA implementation type must continue to be a well-behaving EJB, abstaining from thread and class loader management, and the SCA-enabled Java EE runtime must also continue to behave as in accordance with the EJB container contract.

5.1.8 Use of Implementation Scopes with Session Beans

The lifecycle of a stateless session bean is not impacted by its use in an SCA context. The instance is returned to the free pool as soon as it finishes servicing the request, regardless of whether the call was made over an SCA wire or over using an EJB proxy object. In the terminology provided in [4], a stateless session bean always has a STATELESS implementation scope. An SCA-enabled Java EE runtime is not required to provide means for tuning or customizing this behavior.

Similarly, the lifecycle of a stateful bean is, by default, not impacted by its use in an SCA context. The bean instance remains (modulus passivation/activation cycles) until it times out or one of its @Remove methods are called. In the terminology provided in [4], a stateful session bean has CONVERSATIONAL implementation scope.

5.1.9 SCA Conversational Behavior with Session Beans

The SCA Assembly Specification [3] introduces the concept of *conversational interfaces* for describing service contracts in which the client can rely on conversational state being maintained between calls, and where the conversational identifier is communicated separately from application data (possibly in headers). Note that a conversational contract assumes association with a conversationally scoped implementation instance such as stateful bean. Section 5.1.1 defines how business interfaces are mapped to SCA service. SCA conversational interface must not be used with a stateless bean.

5.1.10 Non-Blocking Service Operations

Service operations defined by a Session Bean's business interface may use the @OneWay annotation to declare that when a client invokes the service operation, the SCA runtime must honor non-blocking semantics as defined by the SCA assembly Specification [3].

5.1.11 Accessing a Callback Service

Session Beans that provide the implementation of SCA components and require a callback service may use @Callback to have a reference to the callback service associated with the current invocation injected on a field or setter method.

5.2 Using Message Driven Beans as Implementation Types

Message Driven Beans are the JavaEE construct for consuming asynchronous messages. Message Driven beans may participate in SCA assembly as the implementation type of a component that does not offer any services, but may be configured or wired from. Message-driven beans cannot be instantiated arbitrarily often due to their association with non SCA-controlled endpoints (typically JMS). Therefore, within the assembly of the application package, a message-driven bean may be used as service component implementation at most once (see also section 3).

5.2.1 Dependency Injection

A message driven bean that is the implementation type of an SCA component may use dependency injection to acquire references to the services wired to the component by the SCA assembly. Dependency injection may also be used to obtain the value of properties or a handle to the component's component context. The following table shows the annotations that may be used to indicate the fields or properties to be injected.

Annotation	Purpose
@ComponentName	Injection of component name

@Context	Injection of SCA context into member variable of service component instance
@Property	Injection of configuration properties from SCA configuration
@Reference	Injection of Service references

A complete description of these annotations, and the values associated with them, is given in the Java Common Annotations and APIs specification [4].

When a message driven bean uses dependency injection, the container MUST inject these references after the bean instance is created, and before any business methods are invoked on the bean instance. If the bean has a PostConstruct interceptor registered, dependency injection MUST occur before the interceptor is called.

See also section 3.1 for an overview over the life cycle handling of SCA-enhanced Java EE components.

5.2.2 The Component Type of an Unaltered Message Driven Bean

Unlike Session Beans, Message Driven Beans do not have business interfaces. Therefore, the component type implied from a message driven bean does not offer any SCA services. The bean may, of course, be accessed indirectly over a binding.jms call to its associated queue, but this is not transparent to the SCA assembly.

The component type of a message driven bean that does not use any SCA annotation and is not accompanied by a component type side file is constructed according to the following algorithm:

1. Remote EJB 3 references MAY translate into an SCA references according to section 3.2.
2. Each Simple-Typed Environment Entry of the session MAY translate into an SCA property according to section 3.2.

5.2.3 Providing additional Component Type data for a Message Driven Bean

Several of the annotations described in [4] influence the implied component type of the session bean (or other Java EE construct). The following table shows the annotations that are relevant in a SCA-enabled Java EE runtime.

Annotation	Purpose
@Property	Adds a property to the implied component type. The type of the property is obtained through introspection.
@Reference	Adds a reference to the implied component type. The interface associated with this wire source is obtained through introspection.

An SCA-enable Java EE runtime MUST observe the specified annotations and use them when generating an implied component type.

5.2.4 Creating SCA Components that use Message Driven Beans as Implementation Types

Since both Message Driven Beans and Session Beans are Enterprise Java Beans, both can be uniquely referenced in an ejb-link. Therefore, no new tag is needed to declare a service component instance that is implemented as a Message Driven Bean: an ***implementation.ejb*** (described in section 5.1.6 above) can be used in both cases.

5.2.5 Limitations on the Use of Message Driven Beans as Component Implementation

A few limitations with respect to use as service component implementation apply to Message Driven Beans:

- A Message-Driven Bean may not be given an implementation scope.
- A Message Driven Bean cannot be used to provide a conversational service. It may, of course, access conversational services.

5.3 Mapping of EJB Transaction Demarcation to SCA Transaction Policies

The EJB programming model supports a concept of container managed transaction handling in which the bean provides class-level or method-level information on transaction demarcation that is observed by the EJB runtime implementation. SCA's policy framework [6] in conjunction with the transaction policies specification [10] defines an extended transaction demarcation model using SCA policy intents.

However, since EJB transaction attributes can be defined on the class as well as on the method-level, the EJB model more fine-granular than SCA's transaction model and a simple mapping to SCA policies is not possible.

For class-level transaction demarcation, the following table illustrates the mapping of EJB transaction attributes to SCA transaction implementation policies:

EJB Transaction Attribute	SCA Transaction Policy, required intents on services	SCA Transaction Policy, required intents on implementations
NOT_SUPPORTED	suspendsTransaction	
REQUIRED	propagatesTransaction	managedTransaction.global
SUPPORTS	propagatesTransaction	managedTransaction.global
REQUIRES_NEW	suspendsTransaction	managedTransaction.global
MANDATORY	propagatesTransaction	managedTransaction.global
NEVER	suspendsTransaction	

Note: in the case of MANDATORY and NEVER demarcations, policy mapping is not completely accurate as these attributes express responsibilities of the EJB container as well as the EJB implementer rather then expressing a requirement on the service consumer (see [8]).

We require that EJB's transaction model stays unchanged by SCA, and an SCA-enabled Java EE runtime MUST adhere to the rules laid out in [8].

5.4 Using Web Modules as Implementation Types

As with Message Driven beans, web modules may participate in SCA assembly as the implementation type of a component that does not offer services, but may be configured or wired from.

5.4.1 Dependency Injection

A web module may use dependency injection to acquire references to the services wired to the component by the SCA assembly. Dependency injection may also be used to obtain the value of properties or a handle to the component context. The following table shows the annotations that may be used to indicate the fields or properties to be injected.

Annotation	Purpose
@ComponentName	Injection of component name
@Context	Injection of SCA context into member variable of service component instance
@Property	Injection of configuration properties from SC configuration
@Reference	Injection of Service references

A complete description of these annotations, and the values associated with them, is given in the Java Common Annotations and APIs specification [4].

Due to the multi-threaded nature of web artifacts, in the case where a Reference Proxy targeted to a conversational interface (such as stetefull session beans) may not behave as expected. SCA-Java EE Runtimes may treat this case as an error. The recommended approach to obtain such reference proxy is via usage of ComponentContext.

Dependency injection of values configured from SCA occurs in exactly those locations that the web container can inject values based on the Java EE configuration. An SCA-enabled Java EE server MUST be able to perform dependency injection on the following artifacts.

Name	Interface or Class
Servlets	javax.servlet.Servlet
Servlet filters	javax.servlet.ServletFilter
Event listeners	javax.servlet.ServletContextListener javax.servlet.ServletContextAttributeListener javax.servlet.ServletRequestListener javax.servlet.ServletRequestAttributeListener javax.servlet.http.HttpSessionListener javax.servlet.http.HttpSessionAttributeListener javax.servlet.http.HttpSessionBindingListener
Taglib tag handlers	javax.servlet.jsp.tagext.JspTag

JavaServer Faces technology-managed beans	Plain Old Java Objects (POJOs)

See also section 3.1 for an overview over the life cycle handling of SCA-enhanced Java EE components.

5.4.2 The Component Type of an Unaltered Web Module

Since it does not offer SCA services the component type of a web module does not contain any SCA services. However, it may contain references and properties.

The component type of a web application that does not use any SCA annotation and is not accompanied by a component type side file is constructed according to the following algorithm:

1. Remote EJB 3 references MAY translate into an SCA references according to section 3.2.
2. Each Simple-Typed Environment Entry of the session MAY translate into an SCA property according to section 3.2.

5.4.3 Providing additional Component Type Data for a Web Application

Several of the annotations described in [4] influence the implied component type of the Web application. The following table shows the annotations that are relevant in a SCA-enabled Java EE runtime.

Annotation	Purpose
@Property	Adds a property to the implied component type. The type of the property is obtained through introspection.
@Reference	Adds a reference to the implied component type. The interface associated with this wire source is obtained through introspection.

An SCA-enable Java EE runtime MUST observe the specified annotations and use them when generating an implied component type. All files where dependency injection may occur (see the table in section 5.4.1) MUST be inspected when generating the implied component type.

A web component can provide additional component type data in the side file

WEB-INF/web.componentType

in the web module archive. Using Web Modules as Implementation Types

5.4.4 Using SCA References from JSPs

JavaServer Pages (JSP) tag libraries define declarative, modular functionality that can be reused by any JSP page. Tag libraries reduce the necessity to embed large amounts of Java code in JSP pages by moving the functionality of the tags into tag implementation classes ([6]).

Following this philosophy, a JSP tag library will be made available to expose SCA components in JSP pages. The following snippet illustrates the use of an SCA reference using the tag library:

```
<%@ taglib uri="http://www.osog.org/sca/sca.tld" prefix="sca" %>
```

```

......

<sca:reference name="service" type="test.MyService"  />

<% service.sayHello(); %>
```

An SCA-enabled Java EE runtime MUST support the SCA JSP tag library by providing implementations of the tag-class and tei-class. The servlet container hosting the webapp will instantiate new instances of the tag-class whenever it comes across the SCA specific tag in a JSP page. The tag-class is responsible for doing dependency injection into the JSP page based on the properties provided to the JSP page. The scope of the object injected is PageContext. APPLICATION_SCOPE in case the the interface is not conversational and PageContext. SESSION_SCOPE in case the interface is statefull. The SCA JSP tag also makes available the given reference with a newly declared scripting variable of the same id.

In order to access SCA configuration from JSP pages, JSP page authors MUST import the SCA tag library provided by the SCA runtime and provide all the properties necessary for dependency injection. The required properties are the name of the reference to be injected, and the type of the field (Service interface class name).

All tag libraries are required to provide a TagLibrary Descriptor (TLD). The information provided by via the tag library descriptors will be used by the web application container to handle processing of tags in the jsp page. The TLD of the SCA tag library is show in the following code box

```
<?xml version = '1.0' encoding = 'ISO-8859-1'?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd">
<taglib version="2.1">

  <tlib-version>1.0</tlib-version>
  <short-name>SCA-JSP</short-name>
  <uri>http://www.osoa.org/sca/sca_jsp.tld</uri>
  <description>A tag library for integrating sca components with jsp
  </description>

  <tag>
    <name>reference</name>
    <tag-class><!—To be provided by the SCA runtime implementation □</tag-class>
    <tei-class><!—To be provided by the SCA runtime implementation □</tei-class>

    <attribute>
      <name>name</name>
      <required>true</required>
      <type>java.lang.String</type>
    </attribute>

    <attribute>
```

```
      <name>type</name>
      <required>true</required>
      <type>java.lang.String</type>
    </attribute>


    <body-content>empty</body-content>

  </tag>

</taglib>
```

5.4.5 Creating SCA Components that Use Web Modules as Implementation Types

The ***implementation.web*** tag can be used to declare a service component that is implemented by the web component. It has the following pseudo-schema.

```
<implementation.web web-uri="<module name>"/>
```

As for message-driven beans, a web component can be configured at most once per assembly of the contribution package.

5.4.6 Limitations on the Use of Web Modules as Component Implementations

Because each module is associated with a unique context root, web modules may be used as service component implementation at most once (see also section 3).

Furthermore, a web module may not be given an implementation scope.

6 SCA-enhanced Java EE Archives

The following sections provide a detailed description of how to make use of SCA concepts within and over the scope of Java EE applications and Java EE modules.

We will use the terms ***SCA-enhanced Java EE application*** when referring to Java EE applications that are composed from a mix of Java EE artifacts as well as SCA artifacts and additional implementation artifacts.

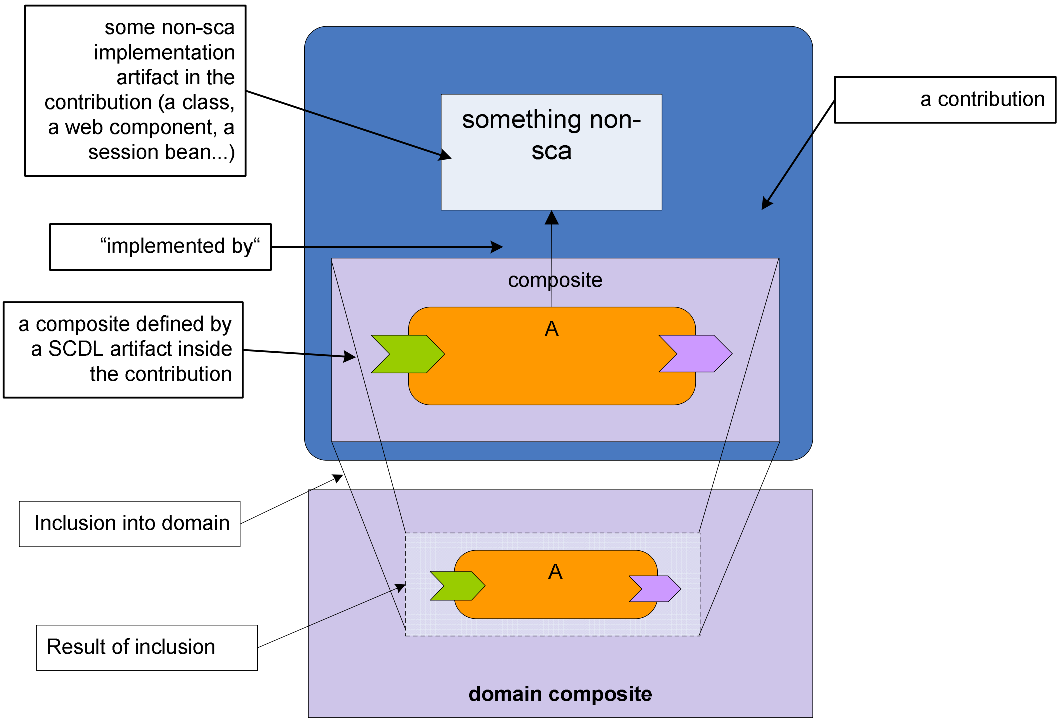
Similarly we will use the term ***SCA-enhanced Java EE module*** for a corresponding construction pertaining to a Java EE module, and we will use the term ***SCA-enhanced Java EE archive*** when referring to either construct.

6.1 Assembly and Deployment of SCA-enhanced Java EE Archives

In this section we will see how to apply SCA assembly concepts when assembling and deploying SCA-enhanced Java EE applications. The SCA assembly specification [3] defines a language and model to make effective use of the implementation types and bindings described in this specification and other specifications (as far as supported by the target runtime environment).

The reader should be familiar with the concepts and terms of the SCA assembly specification [3].

In order to provide a visual representation of assembly and deployment related examples, we use the following graphical notation:


some non-sca implementation artifact in the contribution (a class, a web component, a session bean...)
something non-sca
a contribution
"implemented by"
composite
a composite defined by a SCDL artifact inside the contribution
A
Inclusion into domain
A
Result of inclusion
domain composite

Note: Java EE archives, SCA-enhanced or not, may also be used as service component implementations via the Java EE implementation type. See section 7 for more details.

6.1.1 Java EE Archives as SCA Contributions

A Java EE archive, for example a Java EE application or a Java EE module (a Web application, an ejb module), can be used as an SCA contribution (see [3]).

We will use the term ***Java EE contribution*** for a Java EE archive that is used as an SCA contribution.

A Java EE archive that is being used as an SCA contribution must still be valid according to Java EE requirements, containing all required Java EE artifacts (e.g., META-INF/application.xml in an .ear file).

Many Java EE implementations place some additional requirements on deployable archives, for instance, requiring vendor specific deployment descriptors. A Java EE archive that is an SCA contribution should also fulfill these additional, implementation specific constraints.

As with any regular SCA contribution a Java EE contribution may be associated with a set deployment composites that can be deployed to the SCA domain. A Java EE archive that is being used as an SCA contribution indicates its deployment composites, as well as any imported or exported SCA artifacts, by providing an SCA Contribution Metadata Document at

META-INF/sca-contribution.xml

Section 10.1.2 of the SCA Assembly Specification [3] describes the format and content of this document.

A ***META-INF/sca-contribution-generated.xml*** file may also be present. An SCA-enabled Java EE runtime MUST process these documents, if present, and deploy the indicated composites.

Implementations that support an install step separate from a deployment step may use the add Deployment Composite function (SCA Assembly 1.10.4.2) to allow composites to be added to an installed SCA-enhanced Java EE archive without modifying the archive itself. In this case, the composites will be passed in *by value*. Such feature is useful because it allows the deployer to complete the SCA wiring by adding in the composite.

The deployment of a set of deployment composites from a Java EE contribution, including the exposure of components in the virtual domain composite and of external bindings, takes place *in addition to* Java EE deployment: every Java EE component in the application's deployment descriptors (including EJB3 implied deployment descriptors) will be deployed, whether it is mentioned in a composite or not. See also section 3.1.

Irrespective of how many SCA deployment composites are deployed from a Java EE contribution, only one Java EE deployment will occur.

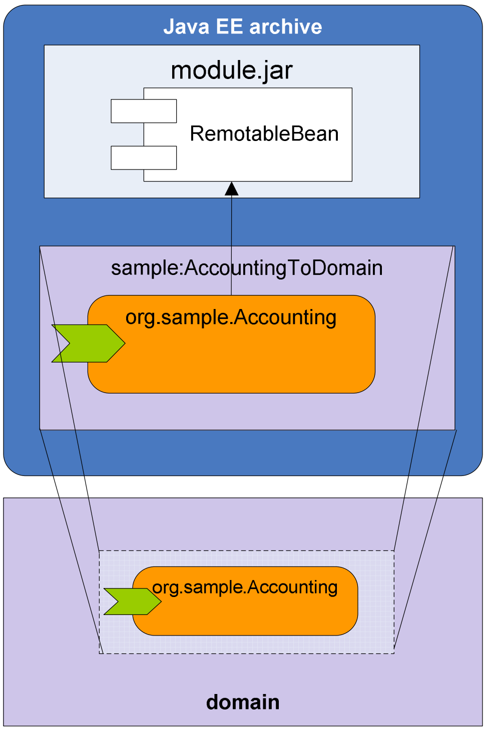
For example, the composite below and the following contribution metadata document would lead to exposure of a contribution of a service component named ***org.sample.Accounting*** to the domain composite. This component exposes a single service AccountReporting that is implemented by the EJB session bean ***module.jar#RemotableBean***, assuming that the session bean ***RemotableBean*** has one business interface by the name ***services.accouting.AccountReporting*** (see also 5.1.2).

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="AccountingToDomain"
                    targetNamespace="http://www.sample.org"
                    xmlns:sample="http://www.sample.org"
                    xmlns="http://www.osoa.org/xmlns/sca/1.0">
```

```

        <component name="org.sample.Accounting">
                <implementation.ejb ejb-link="module.jar#RemotableBean"/>
        </component>
</composite>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:sample="http://www.sample.org">

        <deployable composite="sample:AccountingToDomain"/>
</contribution>
```

Using the diagram notation introduced above we get



While this kind of assembly is very practical for rapidly achieving domain exposure of service components implemented in a Java EE contribution, it provides little encapsulation and information hiding for application level assembly that is not to be exposed in the domain.

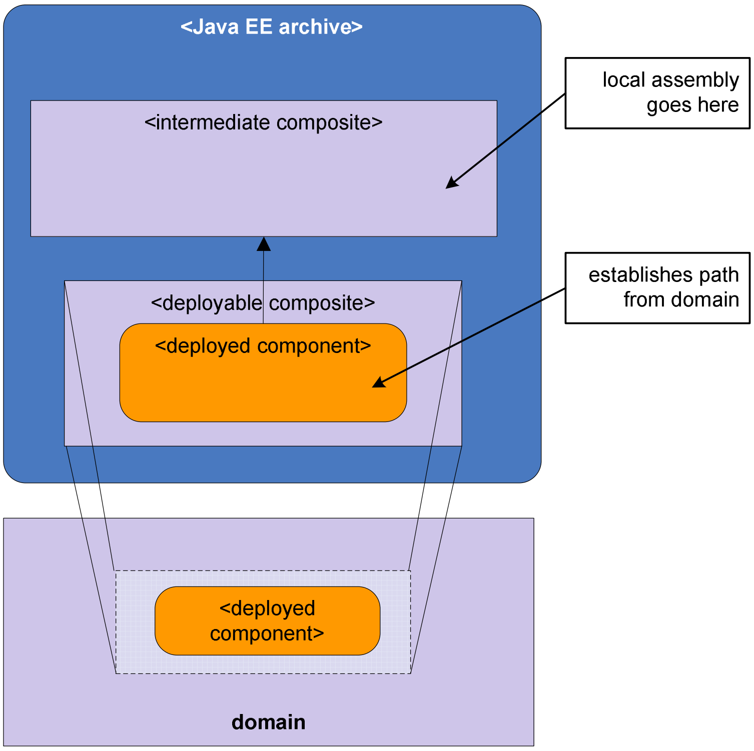
6.1.2 Local Assembly of SCA-enhanced Java EE Applications

On an SCA-enabled Java EE runtime SCA assembly extends Java EE assembly by providing a framework for additional implementation types, bindings, and wiring capabilities. For instance, SCA makes it possible to wire an EJB component to a BPEL process. Such application internal wiring, between standard Java EE components and SCA components whose implementations may not be Java classes (supported implementation and binding types will, of course, vary from implementation to implementation) is a major benefit of SCA.

Users should take advantage of this benefit without requiring explicit contribution of components to the domain and it is often advantageous to separate the application's internal wiring from the components that the application wishes to expose in the domain, in particular, to encapsulate the internal wiring and components.

Nevertheless, consistency with SCA's assembly model requires having a well defined URI path from the domain to any deployed service component.

Therefore, in order to achieve a compliant contribution on the one hand and yet reflect a Java EE archive locally scoped assembly, an application assembler should introduce an intermediate composite that is in turn used as a domain deployed component implementation, as shown in the following abstract construction:



In order to ease the implementation of this typical application assembly approach and in order to provide a developer-friendly, convenient local assembly for SCA-enhanced Java EE applications, SCA enabled Java EE runtimes must support the application composite.

6.1.3 The Application Composite

A Java EE contribution may define a distinguished composite, the ***application composite***, that supports the use of SCA programming model within the scope of the Java EE archive.

The application composite has two particular characteristics:

1. The application composite may be directly or indirectly used as an composite implementation or by inclusion into some deployment composite.
   However, if that is not the case, the SCA implementation MUST logically insert a deployment composite into the archive that contains a single component, named after the application composite, that uses the application composite as its implementation. In addition this deployment composite MUST be deployed into the domain. Consequently the services and references that were promoted from the application composite are exposed into the domain.
2. The application composite supports automatic (logical) inclusion of SCDL definitions that reproduce the component type of the JEE implementation type into the composite's component type. See section 7.2 7.1.3 for a detailed description of the includeDefaults feature.

Application archives (.ear files) that are being used as SCA contributions define the application composite by a composite definition at

META-INF/application.composite

in the enterprise application package. The Java EE specification also supports deployment of single application modules. This method of deployment is particularly popular for web application modules but also used for EJB modules and resource adapter modules. We treat single modules as a simplified application package. The application composite for these archives is defined at

WEB-INF/web.composite

for web modules, and in

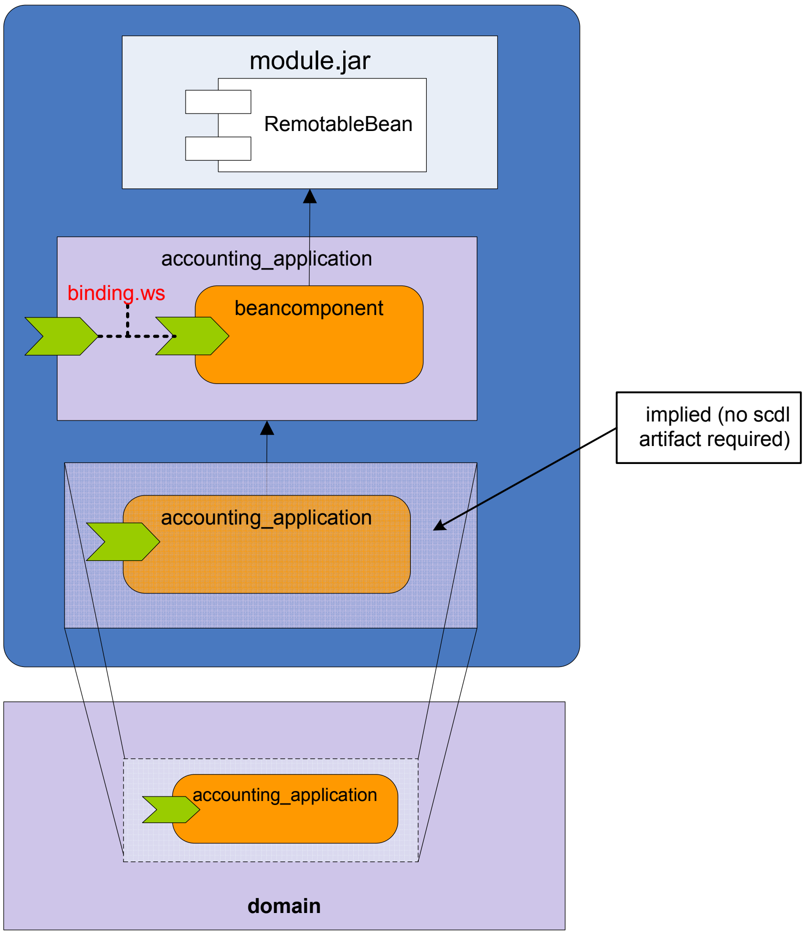
META-INF/ejb-jar.composite

for EJB modules.

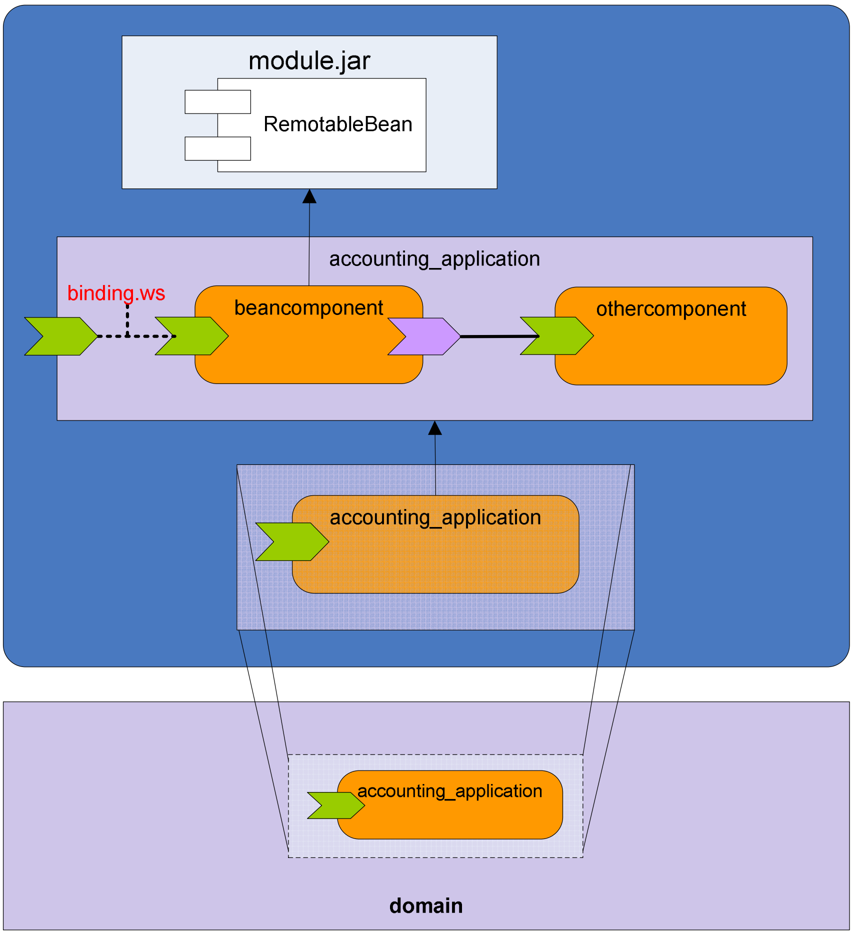
For example the following ***application.composite*** file configures a property of a session bean ***RemotableBean*** and exposes its remote interface service to the domain using a default web service binding.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="accounting_application"
        targetNamespace="http://www.sample.org"
        xmlns="http://www.osoa.org/xmlns/sca/1.0">

        <service name="AccountReporting" promote="beancomponent/AccountServiceRemote">
                <binding.ws/>
        </service>

        <component name="beancomponent">
                <implementation.ejb ejb-link="module.jar#RemotableBean"/>
                <property name="currency">EUR</property>
        </component>
</composite>
```

By definition the application composite implies the generation of a deployment composite that deploys a single component to the domain like this:



The EJB-implemented service component ***beancomponent*** may be modified in a later version so that it makes use of another service component ***othercomponent*** (whose implementation technology we ignore for the sake of the example). It can do so by modifying the application composite but without changing its domain exposure:



6.1.4 Domain Level Assembly of SCA-enhanced Java EE Applications

As applications expose themselves in the SCA domain, they make themselves available for SCA wiring. In this way, SCA allows Java EE applications to do cross application wiring. To illustrate this, we proceed with the example. Another enterprise application, can wire to the provided service by providing a suitable deployment composite. In the example below assume the following contribution metadata document:

```
<?xml version="1.0" encoding="UTF-8"?>
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:here="http://www.acme.com">

        <deployable composite="here:LinkToAccounting"/>
</contribution>
```

Where

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="LinkToAccounting"
                targetNamespace="http://www.acme.com"
                xmlns:here="http://www.acme.com"
```

```
                        xmlns="http://www.osoa.org/xmlns/sca/1.0">

          <component name="com.acme.TicketSystem">
                    <implementation.composite name="here:ticketing_application"/>
                    <reference name="AccountReporting"
                   target="org.sample.Accounting/AccountReporting"/>
          </component>
</composite>
```

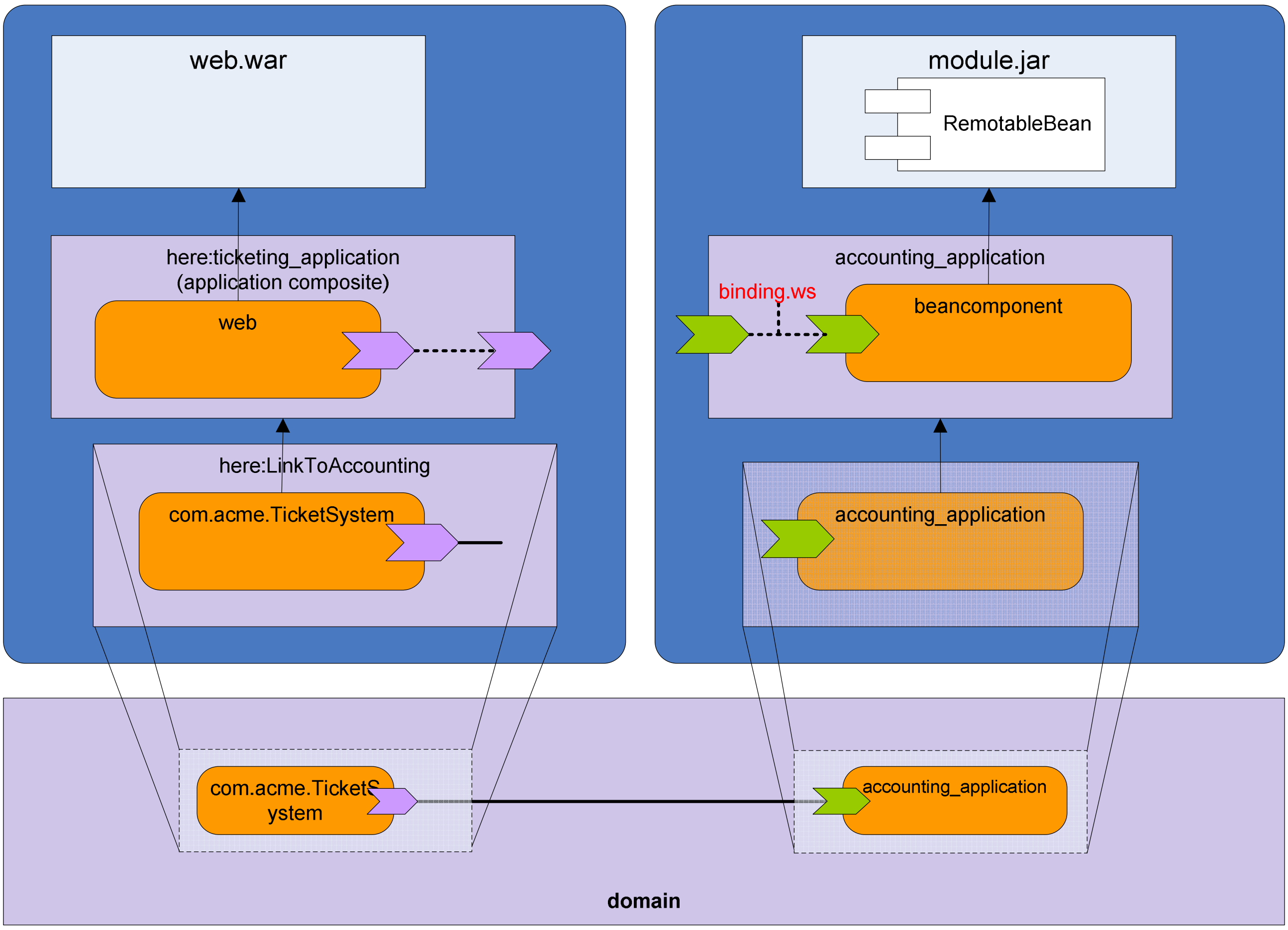
And the application composite is defined as:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="ticketing_application"
          targetNamespace="http://www.acme.com"
          xmlns="http://www.osoa.org/xmlns/sca/1.0">


          <component name="web">
                    <implementation.web web-uri="web.war"/>
          </component>

          <reference name="AccountReporting" promote="web/AccountReporting"/>

</composite>
```

Note that the application composite is used as a component implementation of a composite that is included into the domain. This way, the application composite can participate in domain assembly explicitly (rather than implicitly as demonstrated before).

The example above results in the wiring of a reference ***AccountReporting*** of the web component ***web.war*** to the domain level service ***org.sample.Accounting/AccountReporting.***

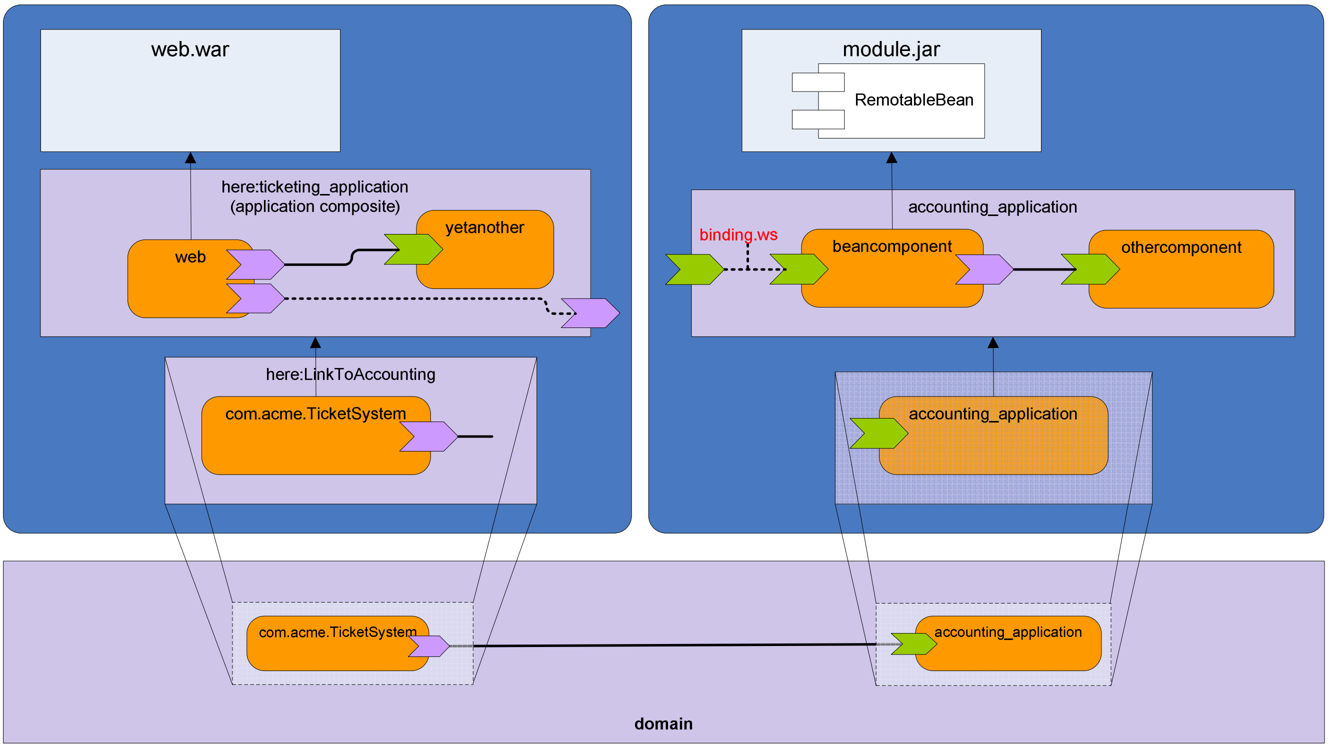
This assembly example has the following graphical representation:



777

778

779 Again, to justify the introduction of an intermediate composite in the contribution on the left hand side,
780 assume the web application was modified to use another local service component ***yetanother***:



Note that the new component could be introduced by a local change of the application composite without affecting the overall assembly.

6.1.5 Import and Export of SCA Artifacts

The import and export of SCA artifacts across contributions for example to be used as composite definitions is described in the assembly specification.

For the specific case of the location attribute of the import element of the ***META-INF/sca-contribution.xml*** document a vendor specific resolution mechanism should be provided.

6.1.6 Resolution of WSDL and XSD artifacts

Composite files and other SCA artifacts may reference, directly or indirectly WSDL and XML Schema documents that are not hosted locally, or which cannot be modified to suit the local the local environment. The OASIS XML Catalogs 1.1 specification [11] defines an entity catalog that can be used to avoid costly remote calls, or to provide a mechanism through which customized versions of docments can be provided without changing application code. Specifically, the XML Catalogs specification provides a mechanism through which

- an external entity's public identifier and/or system identifier can be mapped to a URI reference.
- the URI reference of a resource can be mapped to another URI reference.

Support for the OASIS XML Catalogs 1.1 specification is mandated by JAX-WS, and an SCA-enabled Java EE runtime MUST resolve WSDL and XML Schema artifacts in a manner consistent with JAX-WS.

Specifically, when an SCA-enable Java EE archive is deployed, the process of resolving any URIs that point to WSDL or XML schema documents MUST take into account the catalog that is constructed from all META-INF/jax-ws-catalog.xml found in the archive, and resolve the reference as prescribed in the XML Catalogs 1.1 specification.

7 Java EE Archives as Service Component Implementations

The previous section described how Java EE archives can be represented in SCA where each of the Java EE components in the archive get mapped to separate SCA components. We also allow an alternative formulation, where the entire archive to be represented as a single coarse-grained component within SCA.

The ***JEE implementation type*** supports this use. It has the following pseudo schema:

```
<implementation.jee archive="...">
        <xs:any/>*
</implementation.jee>
```

The ***archive*** attribute specifies a relative path to the Java EE archive that serves as implementation artifact. The context of that relative path (the value ".") is the location of the artifact that contains the ***implementation.jee*** element. All Java EE components contained in the archive will deployed, regardless of any SCA enhancements present (see also section 3.1).

Every deployed SCA component using the JEE implementation type represents a deployment of the referred Java EE archive. Implementers are encouraged to make use of the extensibility of the JEE implementation type declaration to provide deployment plan meta-data as to support vendor-specific deployment features as well as multiple deployments of one Java EE archive.

The archive that is referred to by <implementation.jee> may be an artifact within a larger contribution (i.e. an EAR inside a larger ZIP file), or the archive may itself be a contribution. In the latter case, the @archive attribute can be left unspecified, and the archive will be assumed to be the archive of the contribution itself.

The component type derived from a Java EE archive depends on whether it has been enhanced with SCA artifacts and contains an application composite or not – as described in following sections.

7.1 The Component Type of a non-SCA-enhanced Java EE Archive

Java EE modules, in particular EJB modules and Web modules are frequently designed for re-use in more than one application. In particular EJB session beans provide a means to offer re-usable implementations of business interfaces. In addition Java EE modules can use EJB references as a point of variation to integrate with the assembly of a hosting application.

7.1.1 The Component Type of non-SCA-enhanced EJB Module

The component type of an EJB module, with respect to the JEE implementation type is defined by the following algorithm:

1. Each EJB 3 business interface with unqualified name ***intf*** of a session bean ***bean*** translates into a service by the name ***bean_intf***. The interface of the service and the requirement for EJB intent is derived as in sections 5.1.1 and 5.1.2.

2. Each EJB 3 reference with name ***ref*** of a session bean ***bean*** translates into an SCA reference of name ***bean_ref***. The interface of the reference is derived according to section 3.2. The reference's name may require escaping as defined in section 3.2.

For example, an EJB 3 module ***reusemodule.jar*** may contain a session bean definition ***UsesOthersBean***

```
package com.sample;

import javax.ejb.EJB;
import javax.ejb.Stateless;

@Stateless(name="UsesOthersBean")
public class UsesOthersBean implements UsesOthersLocal {

        @EJB
        private IUOBRefService ref;

        // ...

}
```

that, by use of annotations in this case, has an EJB reference by name ***com.sample.UsesOthersBean/ref*** and the business interface ***IUOBRefService*** (note that alternatively the EJB reference could have been declared in the module's deployment descriptor ***META-INF/ejb-jar.xml***).

When appling ***implementation.jee*** this would result in a component type of the following form:

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
   <service name="UsesOthersBean_UsesOthersLocal">
      <interface.java interface="com.sample.UsesOthersLocal" />
   </service>

   <reference name="UsesOthersBean_com.sample.UsesOthersBean_ref">
      <interface.java interface="com.sample.IUOBRefService" />
   </reference>
</componentType>
```

7.1.2 The Component Type of a non-SCA-enhanced Web Module

As for EJB modules, Web Modules may be re-usable. The component type of a Web module conforming to the Java Servlet Specification Version 2.5 ([6]) is defined as follows:

1 Each EJB 3 reference with name ***ref*** of translates into an SCA reference of name ***ref***. The interface of the reference is derived according to section 3.2. The reference's name may require escaping as defined in section 3.2.

For example, a Web application with the following Servlet

```
package com.sample;

import java.io.IOException;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class ReusableServlet extends javax.servlet.http.HttpServlet implements javax.servlet.Servlet {

        @EJB
        private UsesOthersLocal uobean;

        public void service(ServletRequest req, ServletResponse resp)
        throws ServletException, IOException {
                // ...
        }
}
```

implies the following component type

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
   <reference name="com.sample.ReusableServlet_uobean">
      <interface.java interface="com.sample.UsesOthersLocal" />
   </reference>
</componentType>
```

7.1.3 The Component Type of a non-SCA-enhanced Java EE Application

The component type of a non-SCA-enhanced Java EE application is defined as follows:

Each EJB 3 session bean business interface with unqualified name ***intf*** of a session bean with mapped name ***mname*** translates into a service by the name ***mname_intf.*** The interface of the service is derived as in section 5.1.1. The service name is subject to escaping rules as described in section 3.2.

In the absence of optional extensions, the component type of a non-SCA-enhanced Java EE application does not contain SCA references. However, as an optional extension of the way in which SCA support is provided for Java EE applications, an SCA runtime can choose to provide the capability of re-wiring EJB references using SCA. If an SCA runtime provides this optional extension, then the following rule is applied:

Each EJB 3 remote reference of each session bean within the Java EE application is exposed as an SCA reference. If the remote reference has the name ***ref*** and the name of the session bean is ***beanname***, the SCA reference name is ***beanname_ref***. The reference has an interface derived according to section 3.2. The reference name is subject to the escaping rules as described in section 3.2. Each EJB reference has a target (within the Java EE application) that is the EJB identified by the configuration metadata within the JEE application - it is this target which may be overridden by a new target identified in the SCA metadata of the component using the JEE application. The multiplicity of the generated reference is 0..1. The generated reference must require the “ejb” intent :

```
<intent name="ejb" constrains="sca:binding">

<description> The EJB intent requires that all of the semantics required by the Java EE specification for a communication to or from an EJB must be honored </description>

</intent>
```

This optional extension is in no way required to be provided by any given SCA runtime and that, as a result, it is unadvisable to rely on the capability of rewiring EJB references when porting applications between different runtimes.

7.2 The Component Type of an SCA-enhanced Java EE Archive

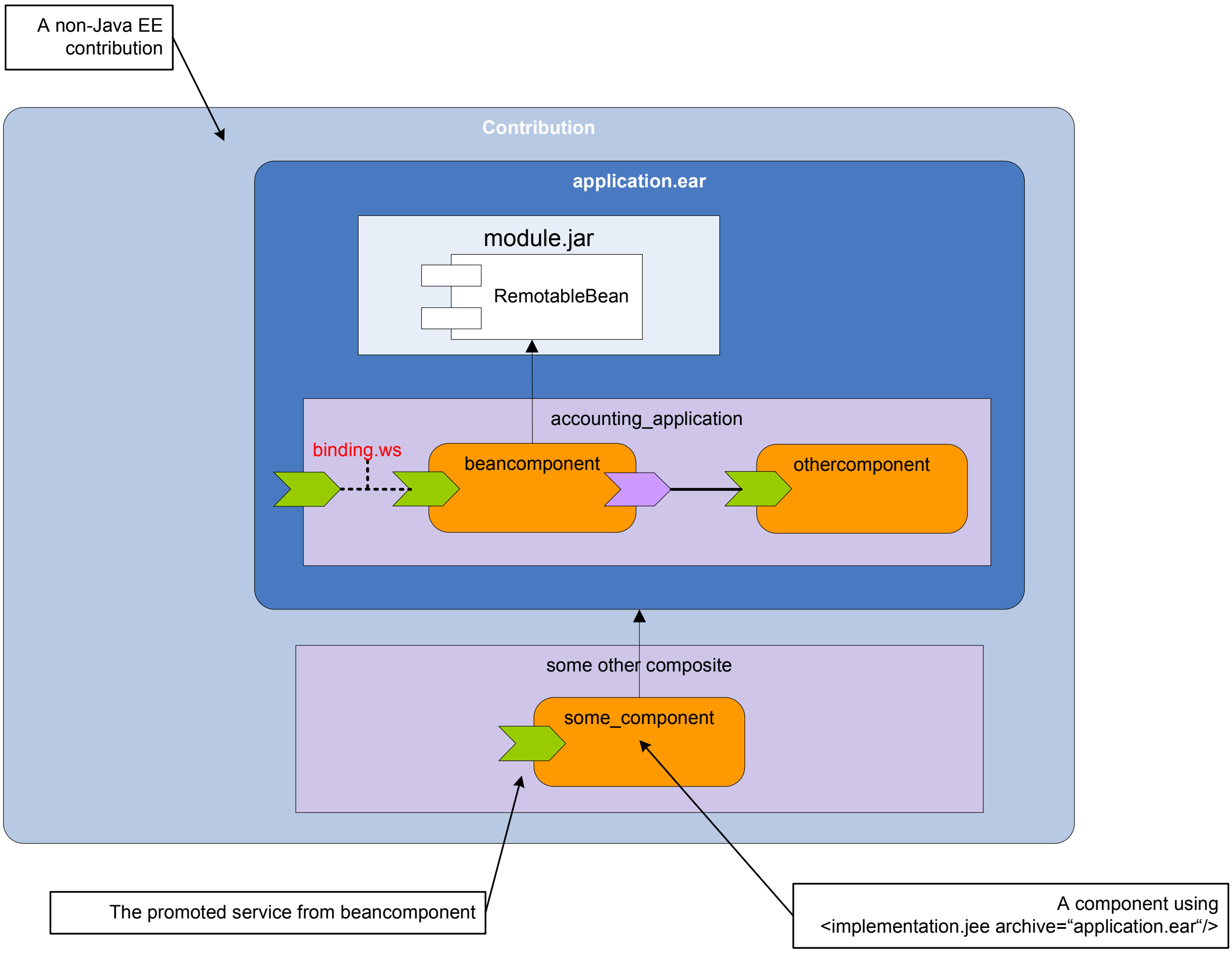
A Java EE archive that contains an application composite (see the section 6.1.3) has the component type of the application composite as its component type when used with the JEE implementation type.

Example: Let’s assume the right hand side application from the example in section Domain Level Assembly of SCA-enhanced Java EE Applications was packaged in an archive ***application.ear*** and would be used as part of a larger non-Java EE contribution that declares a service component in some other composite that uses the archive ***application.ear*** as implementation artifact.

In that case the component type of the EAR archive would expose one service, the ***AccountReporting*** service:

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
        <service name="AccountReporting">
                <binding.ws/>
                <interface.java interface="services.accounting.AccountReporting"/>
        </service>
</componentType>
```

Or, graphically:



This way, the application composite provides fine-grained control over what services, references, and properties are exposed from a Java EE archive.

In cases where a given non-enhanced Java EE archive is already in use as a service component implementation and the need arises to extend it by SCA assembly meta-data, it is desirable to have a smooth and controlled transition from the exposure defined for non-enhanced archives.

That can be achieved using the ***includeDefaults*** attribute that can be specified on composite and component elements. It has the default value "***false***" and is defined in the name space ***http://www.osoa.org/xmlns/sca/1.0/jee***.

Using this attribute on the application composite's composite declaration with a value "***true***" leads to a (logical) inclusion of SCDL definitions into the application composite that reproduce the component type of the Java EE archive as if it was not SCA-enhanced.

For a Java EE application archive, the included SCDL is constructed by the following algorithm:

1. For every EJB or web module that has services or references exposed according to section **Error! Reference source not found.**, a corresponding implementation.ejb or implementation.web

component is included, if that EJB or Web module is not used as a component implementation elsewhere already.

2. For every service or reference that is derived according to section **Error! Reference source not found.**, a composite level service or reference declaration is included, by the same name, promoting the corresponding EJB service or reference.

Corresponding algorithms apply for the case of a standalone Web module (section 7.1.2) and a standalone EJB module (section 7.1.1).

Example (continued): Assume furthermore that the EJB module ***module.jar*** additionally contains the ***AccountServiceImpl*** session bean of section 5.1.2 and the application composite is modified as shown below (note the use of ***includeDefaults***).

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="accounting_application"
        targetNamespace="http://www.sample.org"
        xmlns=http://www.osoa.org/xmlns/sca/1.0
        xmlns:scajee=http://www.osoa.org/xmlns/sca/1.0/jee
        scajee:includeDefaults=”true”
        >

        <service name="AccountReporting" promote="beancomponent/AccountServiceRemote">
                <binding.ws/>
        </service>

        <component name="beancomponent">
                <implementation.ejb ejb-link="module.jar#RemotableBean"/>
                <property name="currency">EUR</property>
        </component>
</composite>
```

That alone would not change the component type of the archive. However, if we additionally assume the session bean ***AccountServiceImpl*** was given a mapped name ***services/accounting/AccountService***, the component type of the EAR archive would expose two services, ***AccountReporting***, ***services_accounting_AccountService_AccountService***.

The logical include to the application composite constructed following the algorithm above is this:

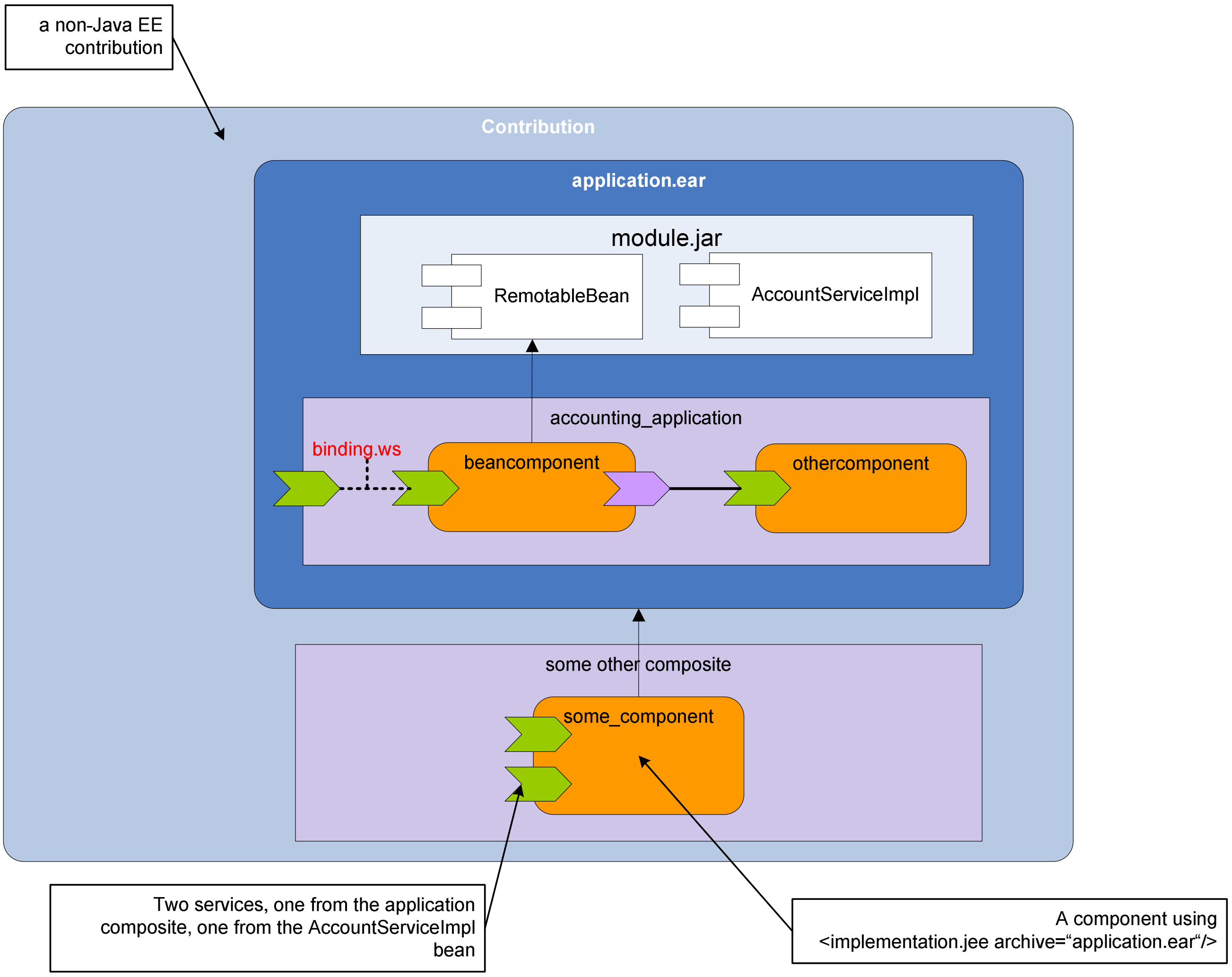
```
<service name="services_accounting_AccountService_AccountService"
        promotes="[some name]/AccountService" />

<component name="[some name]">
        <implementation.ejb ejb-link="module.jar#AccountServiceImpl" />
</component>
```

As a result, we would get the following component type:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
        <service name="AccountReporting">
                <binding.ws/>
        </service>

        <service name="services_accounting_AccountService_AccountService"/>
</componentType>
```

Or, graphically:


a non-Java EE contribution
Contribution
application.ear
module.jar
RemotableBean
AccountServiceImpl
accounting_application
binding.ws
beancomponent
othercomponent
some other composite
some_component
Two services, one from the application composite, one from the AccountServiceImpl bean
A component using <implementation.jee archive="application.ear"/>

The same result can be achieved by declaring the ***includeDefaults*** attribute on a component declaration that uses the ***AccountServiceImpl*** session bean as implementation:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="accounting_application"
        targetNamespace="http://www.sample.org"
        xmlns="http://www.osoa.org/xmlns/sca/1.0"
```

```
    xmlns:scajee="http://www.osoa.org/xmlns/sca/1.0/jee"
>

    <service name="AccountReporting"
        promote="beancomponent/AccountServiceRemote">
        <binding.ws/>
    </service>

    <component name="beancomponent">
        <implementation.ejb ejb-link="module.jar#RemotableBean" />
        <property name="currency">EUR</property>
    </component>

    <component name="accounting" jee:includeDefaults=”true”>
        <implementation.ejb ejb-link="module.jar#AccountServiceImpl"/>
    </component>
</composite>
```

8 References

[1] Java ™ Platform, Enterprise Edition Specification Version 5
http://jcp.org/en/jsr/detail?id=244, http://java.sun.com/javaee/5

[2] SCA EJB Session Bean Binding V1.00
http://www.osoa.org/download/attachments/35/SCA_EJBSessionBeanBinding_V100.pdf

[3] SCA Assembly Model V1.00
http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf

[4] SCA Java Common Annotations and APIs V1.00
http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf

[5] SCA Java Component Implementation V1.00
http://www.osoa.org/download/attachments/35/SCA_JavaComponentImplementation_V100.pdf

[6] SCA Policy Framework V1.00
http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf

[7] Java Servlet Specification Version 2.5
http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html

[8] Enterprise JavaBeans 3.0
http://jcp.org/en/jsr/detail?id=220

[9] SCA JMS Binding V1.00
http://www.osoa.org/download/attachments/35/SCA_JMSBinding_V100.pdf

[10] SCA Transaction Policy Draft V1.00
http://www.osoa.org/download/attachments/35/SCA_TransactionPolicy_V1.0.pdf

[11] Norm Walsh. XML Catalogs 1.1. OASIS Committee Specification, OASIS, July 2005.
http://www.oasis-open.org/committees/download.php/14041/xml-catalogs.html

9 Appendix A – use cases

9.1 Technology Integration

SCA can be used as the scale-out model for Java EE applications, allowing Java EE components to use, be used by, and share a common deployment lifecycle with components implemented in other technologies, for instance, BPEL.

As an example, imagine a sample shop in which the graphic UI is implemented as a servlet or a JSF, the persistence logic is implemented in JPA and exposed using session beans, but the order process is implemented in BPEL. Using standard technologies, the JavaEE components would have to access the BPEL process over its exposed web services. Conversely, in order for the implemented persistence logic to be used from the BPEL process, the session beans must be exposed as web services, typically using JAX-WS.

There are several drawbacks to this approach. Conceptually, the BPEL process is part of the application, however, in the standard deployment described above, the BPEL process is deployed separately from the Java EE application; they do not share life cycle or infrastructure. The use of WebServices as wire protocol imposes other drawbacks. Transaction management and enforcing security policies become much more difficult, and the overhead associated with service invocations increases.

To make the example a bit more concrete, let us imagine that the application's web front-end, implemented as a servlet, will invoke the BPEL process. The BPEL process will, in turn, invoke a session bean called “OrderService”, which uses JPA technology to persiste the order information.

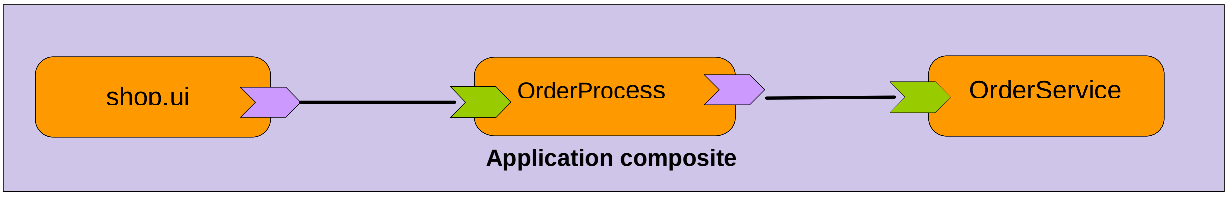
The first step might be to prepare the servlet to make the cross technology call. This is done simply by adding a field with the appropriate business interface, and annotating it with an @Reference tag.

```
public class ControllerServlet extends HttpServlet implements Servlet {
     @Reference protected IOrderProcess orderProcess;
          …
     protected void service(HttpServletRequest request,
               HttpServletResponse response) throws Exception {
     …
     orderProcess.placeOrder(orderData);
     …
}
```

Such a snippet should be familiar to anyone who has used the EJB client model. The main difference between the @EJB and the @Reference annotation is that @EJB tells the user which technology is being used to implement the service, whereas @Reference leaves this undetermined.

The next step in creating a cross technology application in SCA is to create the assembly file that hooks together our components, and links each to an implementation. In this case, there are three SCA components: the web front-end, the BPEL component, and the EJB that offers the persistence service. Note that there may be many more EJBs and web components in our Java EE application, we do not need to represent them all as SCA components. Only those Java EE components that will be wired to or from, or otherwise configured from SCA, need to be represented in the SCA assembly.

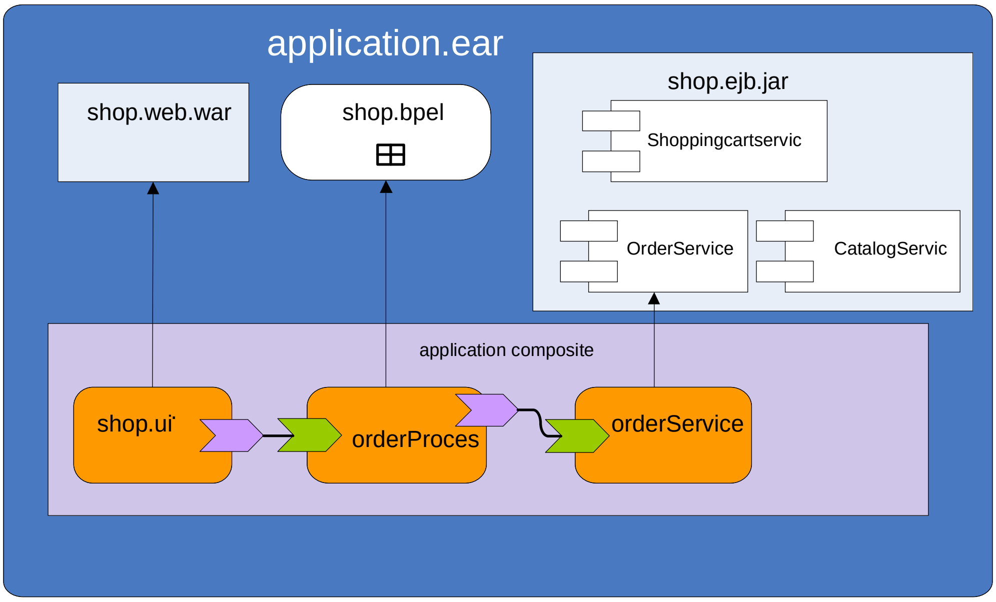
The following figure shows how the components are hooked together.



The composite file looks like this:

```
<sca:component name="OrderService">
    <sca:implementation.ejb ejb-link="shop.ejb.jar#OrderService"/>
    <sca:service name="IOrderService">
      <sca:interface.java
           interface="sample.shop.services.IOrderService"/>
    </sca:service>
</sca:component>
<sca:component name="shop.ui>
    <sca:implementation.web web-uri="shop.web.war"/>
    <sca:reference name="orderProcess" target="OrderProcess"/>
</sca:component>
<sca:component name="OrderProcess">
    <sca:implementation.bpel process="shop.bpel" version="2.0"/>
    <sca:reference name="orderServicePL" target="OrderService">
    <sca:service name="OrderProcessRole"/>
</sca:component>
```

There are several ways in which such a cross-technology application could be deployed. If we consider the BPEL process to be part of the application, conceptually on the same level as the application web or EJB components, then it makes sense to deploy the cross technology application as an ***SCA-enhanced Java EE archive,*** that is, the SCA and BPEL artifacts are packed into the EAR file. The following figure depicts the contents of this the enhanced archive.



The advantage of deploying an SCA-enhanced Java EE archive is that we can leverage the tooling, monitoring and application lifecycle management capability already present on the Java EE server.

9.2 Extensibility for Java EE Applications

SCA \ Java EE can be used for the following problem -- a company (let's call it ACME) wishes to provide a Java EE application to its customer so that the customer can integrate this application into its own environment. Ideally the application should have some predefined "extension points" which would allow the customer to hook its own implementations over the default one. For example the customer may wish to override some specific logic provided by the company acme in an EJB and instead introduce its own existing functionality written in some proprietary non-Java programming model or via some of the predefined SCA possibilities (another EJB, JMS, WS call, etc.)

Here it is assumed, that the company ACME will predefine explicitly some extension points, another possible use case that optionally some SCA runtimes may support is to allow each remote ejb reference to be reconfigured , please see section - 7.1.3 (The Component Type of a non-SCA-enhanced Java EE Application) for more information.

The exposure of the extension point by the ACME company can be done in several way - fine grained approach using implementation.ejb as in section 5.1 or using implementation.jee as in section 7, by explicit usage of componentType side files or by exposing extension points via the @Reference annotation, via usage of application.composite with includeDefaults or via usage of other composite definitions.

Here it is demonstrated just one such approach :

The EJB from ACME would look like

```
package com.acme.extensibility.sample;
import javax.ejb.Stateless;
import org.osoa.sca.annotations.Reference;


@Stateless(name=" ACMEBean ")
public class BaseBean implements BaseLocal {
```

A default value for the fields would be the EJB as defined by the Java EE specs, however by usage of @Reference, it is indicated that it is possible via using SCA to override that and inject a proxy capable of transferring the request according to the SCA rules.

```
private @Reference @EJB com.acme.extensibility.ExtensionInterface
extensionPoint;

public void  businessLogic() {
        extensionPoint.doSomething();
}
```

In order to contribute to the SCA domain and expose the reference, the ACME company has put the following two artifacts in the META-INF directory of the EAR :

```
<?xml version="1.0" encoding="UTF-8"?>
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:acme="http://www.acme.com.org">
        <deployable composite="acme:AcmeCompositeName"/>
</contribution>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="AcmeCompositeName"
        targetNamespace="http://www.acme.com"
        xmlns:acme="http://www.acme.com.org"
        xmlns="http://www.osoa.org/xmlns/sca/1.0">

        <component name="ACME_component ">
                <implementation.ejb ejb-link="ACMEJAR.jar#ACMEBean "/>
                <reference name="extensionPoint">
                   <interface.java interface="com.acme.extensibility.ExtensionInterface"/>
                </reference>
        </component>
</composite>
```

After exposing the extension point in such way and delivering the EAR to the customer, the customer can wire to it via SCA to its own non-Java technology xyz. The following contribution to the domain demonstrates how this can be done...

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="CompositeName"
   targetNamespace="http://www.org.customer.foo"
   xmlns:customer="http://www.org.customer.foo"
   xmlns="http://www.osoa.org/xmlns/sca/1.0">

  <component name="CustomerCode">
    <implementation.xyz  attribute="someDataForXyz"/>
    <service name="ExtensionTarget">
      <interface.java interface="com.acme.extensibility.ExtensionInterface"/>
    </service>
  </component>
  <wire source="ACME_component/extensionPoint" target="CustomerCode/ExtensionTarget"/>
</composite>
```

10 Appendix B – Support for SCA Annotations

The following table provides information whether SCA annotations are supported in EJB classes or session bean interfaces. Some of the annotations defined in [4] are redundant to Java EE annotations and concepts. These are labelled as "May be supported", it is expected for SCA runtimes supporting these annotations to detect impossible combinations that violate the Java EE specifcations and reject such deployments. Other annotations are labeled as “may be supported” because they represent optional features.

AllowsPassByReference	May be supported	This is a hint to the runtime, which can be disregarded
Callback	Must be supported	
ComponentName	Must be supported	
Constructor	NOT supported	There are no constructors in EJB
Context	Must be supported	
Conversational	Must be supported	Each interface of statefull EJB is treated as it has @Conversational, so the annotation is redundant. In case of stateless EJB-s the stateless semantics still remains, please see the comment for conversationID
ConversationAttributes	May be supported	Providing ways to control the expiration of statefull EJBs by maxAge, maxIdleTime
ConversationID	Must be supported for stateful May be supported for stateless	If there is @Conversational on the interface of stateless bean, the conversationID will be generated by the runtime and may be inserted, the stateless semantic will still be in effect
Destroy	May be supported	Equivalent to @PreDestroy in EJB

EagerInit	NOT supported	There is no composite scope, it has no meaning
EndsConversation	May be supported	Methods that are marked @Remove should be treated as if the corresponding interface method is marked @EndsConversation. Interface methods marked @EndsConversation MUST have corresponding implementation methods marked @Remove.
Init	May be supported	Equivalent to @postConstruct in EJB
Authentication , Confidentiality, Integrity , Itent, PolicySets, Requires	Must be supported on fields already annotated with @reference May be supported on class, session bean interface or on field annotated with @EJB	
Intent, Qualifier	NOT supported	Not relevant, new annotations cannot be defined via EJB
OneWay	Must be supported on fields already annotated with @reference Must be supported as an annotation on interface methods. Must not be supported on class, session bean interface or on field annotated with @EJB	There are async call in EJB 3.1
Property	Must be supported	
Reference	Must be supported	
Remotable	May be supported	Redundant to @Remote.
Scope	May be supported	@Stateless and @Stateful are mappings of stateless, and conversational scopes.
Service	May be supported	

11 Appendix C – schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
            elementFormDefault="qualified">

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

      <xs:element name="implementation.ejb" type="EJBImplementation"
substitutionGroup="implementation"/>
      <xs:complexType name="EJBImplementation">
            <xs:complexContent>
                  <xs:extension base="Implementation">
                        <xs:sequence>
                              <xs:any namespace="##other" processContents="lax"
minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="ejb-link" type="xs:string"
use="required"/>
                        <xs:anyAttribute namespace="##any" processContents="lax"/>
                  </xs:extension>
            </xs:complexContent>
      </xs:complexType>
      <xs:element name="implementation.web" type="WebImplementation"
substitutionGroup="implementation"/>
      <xs:complexType name="WebImplementation">
            <xs:complexContent>
                  <xs:extension base="Implementation">
                        <xs:sequence>
                              <xs:any namespace="##other" processContents="lax"
minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="web-uri" type="xs:string"
use="required"/>
                        <xs:anyAttribute namespace="##any" processContents="lax"/>
                  </xs:extension>
            </xs:complexContent>
      </xs:complexType>
      <xs:element name="implementation.jee" type="JEEImplementation"
substitutionGroup="implementation"/>
      <xs:complexType name="JEEImplementation">
            <xs:complexContent>
                  <xs:extension base="Implementation">
                        <xs:sequence>
                              <xs:any namespace="##other" processContents="lax"
minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="archive" type="xs:string"
use="required"/>
                        <xs:anyAttribute namespace="##any" processContents="lax"/>
                  </xs:extension>
            </xs:complexContent>
      </xs:complexType>
</xs:schema>
```