University of West Bohemia
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

The CoSi Component Model

Specification of CoSi version 2

Přemysl Brada, with Břetislav Wajtr and Vojtěch Liška

Technical Report No. DCSE/TR-2008-07
July 2008

Distribution: public

Table of Contents

1 Introduction

Since the advent of component-based software engineering (CBSE), many component frameworks have been created and used in application development. Enterprise JavaBeans [EJB3], CORBA Component Model [CCM4], Spring [SF25], and OSGi [OR4] serve as good examples of the industrial applicability of component principles.

The universally accepted foundational works [Szy02][Bac00] list several constituting characteristics of components, of which the key one is the need to treat components as opaque black boxes with explicit interface declaration. These characteristics are enforced and the structure of the component's surface[1] is defined by a *component model* (more precisely, its meta-model part), which defines the (functional) features it provides for client components or declares as dependencies, as well as its behavioural specifications and extra-functional properties. The other key roles [Hei01] of component models are to describe the allowed ways of inter-component bindings, i.e. architectural constraints, and aspects like component lifecycle management or capabilities of an underlying runtime framework.

1.1 Goal of this specification

Despite the widespread adoption of the component paradigm (or maybe just the component buzzword), the findings about the fundamental properties of component models which we summarize in a different report [Bra] are not too encouraging, especially for the industrial ones. We have therefore designed an experimental component model that aims to take the best of both worlds – strictly adhering to the fundamental concepts yet providing sufficient practicality.

The model is called *CoSi,* an acronym from *Components Simplified.* In this report we give the full specification of its component model and important aspects of the underlying runtime infrastructure, the container.

The ideas that were driving the design of the CoSi component model are as follows, roughly in decreasing order of importance. In some architectural and practical design decisions the model follows the ideas of the OSGi core, which we think in principle strikes a good balance between (potential) rigour and simplicity, despite the shortcomings described above. In fact, CoSi could be seen as an attempt to build an OSGi-like component model which is formally strong from the black-box and surface representation perspectives.

Strong pure black-box.

Since we believe the black-box concept is at the core of component based software engineering, we want nothing to be acccessible from outside a component except what is explicitly declared as such.

Complete yet minimal feature specification.

The component specification has to contain all basic information about features (existence, name, type), introspection can be used to augment the necessary details.

Maximum simplicity in the underlying infrastructure

That's where the ``simplified'' part of the model's name comes from. The emphasis is on the

1 We use this term instead of the commonly used ``interface'' to avoid mistaking it for an interface type as in Java or IDL; a [SOFA] synonym is *frame.*

component model properties, not on the framework capabilities. In practice this means no distribution, remoting, security, or dynamic updates, simple runtime framework, preference of text over XML formats. OSGi was a strong inspiration in this respect.

Support weakly typed languages.

The component model is designed so that it enables research in suitability of scripting languages for component implementation especially in the context of component substitutability. The Groovy scripting language was chosen for component implementation, for its close ties to Java.

Reasonable feature set.

We want to include practically useful features like events and streams, and use named features; CORBA Component Model was inspirative in this respect.

1.2 Structure of the document

This document is divided into two key parts. In the following chapter we provide the key characteristics that we want the CoSi component model to have. Chapter 3 contains the complete specification of the CoSi component model, framework properties and core services, and the associated run-time interfaces and structures.

2 Requirements

The analysis of current component models and the aims described in the Introduction can be translated into a set of requirements which the new component model has to fulfill. In this section they are listed and described in detail.

2.1 Fundamental properties

The following requirements relate to the core properties of the component model as such.

Flat component model

Component composing is not supported by the framework. All components communicate on the same level and have same access to the container and the container has same level of access to all components.

Pure Black-box

All that bundle provides to it's environment is specified outside bundle implementation. Bundle implementation is not interesting (and not accessible) for other bundles and for container. Consequently, any and all attempts to export, query, bind, or invoke a bundle feature that is not defined in the bundle specification, and even to interrogate its existence, shall be considered as violations of bundle's encapsulation barrier and must fail.

Possibility to provide multiple interfaces with the same type as different services

If a bundle provides one service interface, but contains two (or more) different implementations of this interface, it is necessary to identify these implementations as different services and to provide another service identifier (for example, “a:ServiceInterface” and “b:ServiceInterface”).

Possibility to control the bundle life cycle

The component shall have attached a control interface. This has two roles. First, it enables to control component instance lifecycle. Second, it enables the run-time framework to provide broad possibilities to obtain component meta-data – about attributes, interfaces, inputs/outputs, etc. (Most of this information is specified in the bundle manifest file). Analogous to EJB Home/Controller interface.

2.2 Pragmatic and Framework Properties

The following requirements relate to the goal of simplified development of the run-time framework for the core component model, as well as of the components themselves.

In-process

The complete run-time environment, that is the container and the components it manages, runs in one process within one virtual machine. No distributed component calling is present in framework, to reduce the need to handle data serialization and inter-process communication.

Communication with (external) clients/users through console

Only simple command-line interface will be available for user interaction with container, at least as part of the core framework. (Rich GUI or web-based communication is not precluded but neither is it considered important.)

Framework provides StdIn/StdOut or HtmlIn/HtmlOut interfaces

Bundle implementation provides possibility to redirect input/output to resources provided by the container. In case of HTML communication, framework provides possibility to access a bundle through URL.

Java and Groovy combination

The container core is written in Java. Bundles can be written in Java or Groovy. Using scripting language based on Java provides dynamic typing possibility. Working with bundles written in Groovy is one of the main framework requirements.

3 CoSi Specification

CoSi (Components Simplified) is formal specification of a general Java framework, whose main goal is to provide development, runtime and cooperation of independent software units (bundles). This specification results mainly from OSGi specification [OR4]. Some OSGi elements were taken over, some were modified and some are left out completely see Chapter 7 – differences from OSGi.

The specification consists of:

- *bundle description* - what is a bundle, what does it contain, how to write a bundle and how can a bundle communicate with other bundles
- *container description* - container is a runtime for bundles. Bundle life cycle takes place inside the container. All available pieces of data about bundle are accessible through the container. A bundle can obtain references to services using the container.
- *basic services description* – the container implementation has to provide two basic services for components, one of which represents simple container API, second one represents event-based communication service.

These parts together are called *the framework*. It allows to create applications based on independent bundle composition and provides means for installation, resolving, starting and uninstalling bundles, if they are no longer necessary. Using and resolving bundles is dynamic – you can install bundles at runtime.

3.1 Component model

This section provides specification of the core of CoSi: its component model. We describe the bundle as such, its constituent classes, meta-data and distribution format.

3.1.1 Overview: the CoSi Meta-Model

The CoSi component model contains the entities described in Figure 1 below.

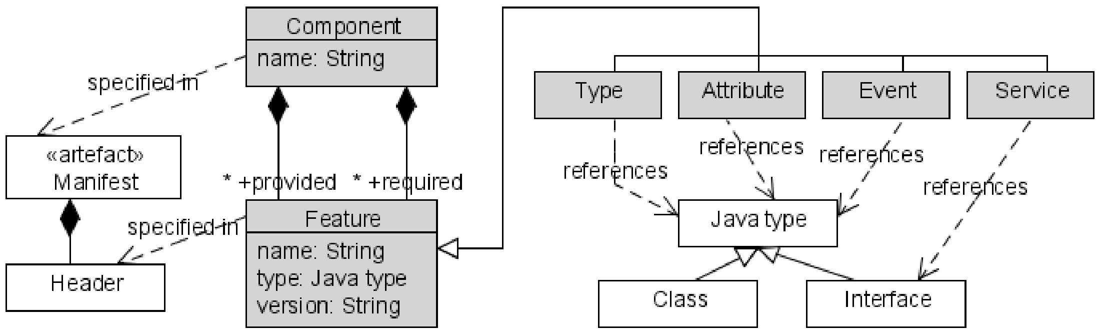

Figure 1: The meta-model of CoSi

The component has a provider, name and version which together provide its unique identification (only the name is shown in Figure 1). Four types of features can be provided and/or required by a component. Each feature has a name (where applicable), type and optional attributes like version identification. Several features of the same type can be specified, distinguished by their names.

Service is an implementation of functionality, specified by a Java interface. Provided services are registered with the runtime container which then mediates the bindings to the requiring components. The binding is realized by service reference objects.

Type referes to a language class or interface. Provided types are exported by the component's packages, and are accessible by the requiring components via the exporter's classloader.

Events enable messaging among components, mediated by a system message service. Events are named and typed, which enables event consumers to set filters on the kinds of events they subscribe to. Both asynchronous and synchronous event delivery is supported.

Attributes define typed values which can be set or read by the component. Attributes can be read-only and read-write, and are implemented as *(key:String, value:Object)* pairs accessible via a system-wide attribute registry.

3.1.2 Bundle contents

In the following text we describe the contents of a bundle archive and means of bundle communication with the container and with other bundles.

Because we treat bundle as a black-box (without implicit access), it has to contain (in addition to class files) so called *bundle manifest* describing the contents of the JAR file and providing information about the bundle. This file uses *headers* to specify information about bundle name, version, dependencies, classpath, etc. The container reads this file during the installation of a bundle and, based on these *metadata,* integrates bundle to the system, links bundle with other bundles, etc.

The manifest file contains descriptive information about the bundle such as:

- **bundle activator** – container uses this activator to start or stop the bundle;
- **interfaces and services** – provided interfaces or services;
- **types** – the bundle can provide some of its inner classes (type) or use other bundle's classes;
- **events** – the bundle specifies which messages will it send or be able to receive;
- **attributes** – bundle can set system variable, which other bundles can read.

3.1.2.1 Bundle activator

Every bundle in CoSi framework is considered *runnable*. It means that one of the bundle classes has to implement `BundleControl` interface (see 4.2.1), which includes `start()` and `stop()` methods. Every bundle integrated to the system is started and stopped using these methods.

`Control-Class` header specifies, which class from inside the bundle implements `BundleControl` interface. `Control-Class` is one of the mandatory headers. Container will refuse to install bundle without one of the mandatory headers.

3.1.2.2 Provided and required services (interfaces)

After the bundle starts, it may want to communicate with other bundles in the system, share information, use their services or provide own services to them.

One way of bundle communication are *services*. The service (in CoSi) is part of the bundle that is made public to its environment and provides functionality. System of services is tightly connected with bundle life cycle. The service is Java object registered as an *interface*. Services are maintained by the *service registry.* Bundles can register their services with the service registry or search the registry for services they want to use.

In CoSi specification, bundles are built around a set of cooperating services, available from the service registry. The service is defined by its *interface* and implemented as *service instance.* The service instance is owned by the bundle that registered service and runs inside that bundle. The bundle has to register the service instance using the service register. This is done to make service sharing controlled by the container.

Dependencies between service-providing bundles and service-using bundles are managed by the container too. For example, the container prevents the bundle from stopping if there are other bundles present in the system that use bundle's services. Container also prevents bundle from installing, if it specifies the need for specific service in the manifest, but there is no such service registered with the bundle.

The framework keeps information about services in the service registry. It provides simple way to search and find services required by specific bundle.

ServiceReference

Each service has an `ServiceReference` type object associated in the service register. This object contains service properties and other service meta-data. Other bundles use this object to get information about registered service or to choose a service that suits their needs the best. This prevents unnecessary bindings between bundles when bundle only needs information about the service, but not the service instance. `ServiceReference` type object can be saved and passed to bundles. If a bundle wants to use the service, on which `ServiceReference` object points, it can ask for the service instance by calling `BundleContext.getService(ServiceReference).`

Service interface

Service interface is the service public methods specification. Bundle programmers create service instances from objects that implement this interface and registers these objects with the service register. Service instance methods are accessible by calling service interface methods a soon as service instance is registered with the container. Particular service instance type is hidden and bundles have access only to it's interface. Service interface type (physical) must be made public, so that bundles using this service could work with the service.

Exported services must be specified in the manifest in `Provide-Interfaces` header. Services required by bundle (imported) must be specified in the manifest in `Require-Interfaces` header. For details about these headers see Appendix A Manifest File Syntax.

3.1.2.3 Provided and required types

For complex services often is not enough only to publish interface type, but it is necessary to publish even supplemental types. Imagine the situation, when a bundle provides a service to fetch data from a database. The service includes methods for database querying and these methods have special return type that contains data from the database. This type is the part of the bundle and also part of the service and in order to use this service, other components need access to this type.

Provided (exported) or required (imported) types are specified in the bundle manifest. Provided types are specified in `Provide-Types` header whilst imported types are available in `Require-Types` header. The container ensures that the bundle will be installed into the system only if there are bundles exporting bundle's imported types (according to the manifest) available and installed.

3.1.2.4 Provided and required events

Another type of communication between bundles is *the event system*. The framework provides standard service `MessageService` (see 4.3.2) that takes care of sending and receiving events (messages) to other bundles.

The `Message` is common Java object inherited from the `Message` abstract class, provided by the container. The bundle has to import `MessageService` service and `Message` type in order to use the event system. The message is identified by its name and type. Message name is simple string that describes the message. For example "`cosi.message.Example`". Message type is the name of the class implementing Message abstract class. The message also contains *properties* – list of pairs of strings `key-value` that puts information about the message more precisely. Message programmer can insert any information into these properties so that receivers could process the message more accurately.

In order for a bundle to send a message, message type and name must be specified in the bundle manifest (`Generate-Events` header) and the bundle has to create the message instance during the runtime, acquire `MessageService` service instance and send the message using `MessageService.sendMessage()` or `messageService.postMessage()` method. The message service ensures that the message will be delivered to all recipients. The bundle sending a message has no information about who receives the message – it is `MessageService` service responsibility. Therefor, message sending is undefinable. That means the sending bundle has no means to specify particular bundle for which the message is intended. If the programmer needs such linkage between bundles, he should choose more describing message name or another means of communication between components (for ex. the service).

There is a difference between `sendMessage()` and `postMessage()` methods in the way of delivering the message. The `sendMessage()` method is *blocking* – method call won't end until all recipients receive and process the message. The `postMessage()` method is *asynchronous* – method call only passes the message to the `MessageService` service and return from the method occurs immediately after passing the message.

In order for a bundle to receive message, it has to register the message *receiver* with the `MessageService` service. The message receiver is a bundle class implementing the `MessageConsumer` interface provided by the container. This receiver contains `receiveMessage(Message)` method that is called by `MessageService` during message sending. A bundle has to specify what kind of messages is it able to receive in its manifest file (`Consume-Events` header). Message name is specified in this header together with its type. Simultaneously `MessageConsumer` interface includes `getAcceptedMessages()` method in which programmer specifies message names that this receiver accepts.

3.1.3 Bundle specification – the Manifest.mf file

In this chapter we will describe fundamentals of bundle manifest headers. For accurate syntax of all headers see Appendix A Manifest File Syntax.

Bundle manifest consist of multiple *headers*. Each header has its own name and value, separated by colon. Header example:

```
Bundle-Version: 1.2.2.build01
```

"`Bundle-Version`" is the header name, "`1.2.2.build01`" is the header value. Version or version interval definitions often appear in headers. These definitions have some specifics. First, we define how are these elements written in the manifest:

3.1.3.1 Version

Version consists of major, minor and micro number, separated by dots. Fourth optional part of version is qualifier, which is version text complement stated after micro number, separated by a dot. Version mustn't contain any white space. Default value for version is `0.0.0`. Common Java method `String.compareTo()` is used when comparing two versions (it means that `"1.1.1.build-05" < "1.1.1.build-1"` is evaluated as true).

3.1.3.2 Version Range

Interval version can appear in headers concerning required types or interfaces. If version-range expression is defined as a single version, it must be interpreted as a version range `[version, ∞)`. If `version-range` is not specified, default value `0` is used. This value maps to `[0.0.0,∞)`. Because `version-range` contains a dash and it could collide with other definitions, `version-range` element must be enclosed with quotation marks. Example definitions of required interfaces:

Examples:

Example	Predicate
`[1.2.3, 4.5.6)`	`1.2.3 <= x < 4.5.6`
`[1.2.3, 4.5.6]`	`1.2.3 <= x <= 4.5.6`
`(1.2.3, 4.5.6)`	`1.2.3 < x < 4.5.6`
`(1.2.3, 4.5.6]`	`1.2.3 < x <= 4.5.6`
`1.2.3`	`1.2.3 <= x`

Following text will show descriptions of headers that may appear in the manifest.

3.1.3.3 Control-Class

Header specifies the bundle activator. It is the name of the class from a bundle that implements `BundleControl` interface. This header is mandatory.

Example:

```
  Control-Class: cz.zcu.fav.kiv.systemshell.impl.BundleActivator
```

3.1.3.4 Bundle-Name

`Bundle-name` header represents bundle name.

Example:

```
  Bundle-Name: com.acme.foo
```

3.1.3.5 Bundle-Version

This header defines the bundle version.

Example:

```
    Bundle-Version: 22.3.58.build-345678
```

3.1.3.6 Require-Services (synonym: Require-Interfaces)

Required service interfaces definition. Developer can specify other parameters that are used when searching for particular interface. The following attributes are supported:

- `versionrange` – Version range that exported interface must have. Syntax corresponds with `version-range` expression. If no value is specified, default value `[0.0.0, ∞)` is used.
- `name` – Name of the exporting service. If there are more services implementing same interface in the system, they are distinguished by this name.
- `bundle-name` – Symbolic name of exporting bundle. DEPRECATED
- `bundle-provider` – Name of bundle creator. DEPRECATED
- `bundle-versionrange` – Version range that must exported bundle have to be chosen. If no value is specified, default value `[0.0.0, ∞)` is used. DEPRECATED

Example:

```
Require-Services:com.acme.Foo;com.acme.Bar; versionrange="[1.23,1.24]",
                    cz.zcu.fav.SomeInterface; versionrange="1.2.1";
                   bundle-versionrange="[1.2, 2.0)"
```

3.1.3.7 Provide-Services (synonym: Provide-Interfaces)

Provided service interfaces definition. Parameters that can be specified for particular provided interfaces. These attributes may or may not be specified.

- `version` – Version of provided interface. This value is different than bundle version!
- `name` – Name of provided service. Bundle can provide multiple services with the same interface. These services are distinguished by this name inside the framework. If name is specified, it mustn't collide with other service implementing the same interface.

The framework assigns following attributes to every providing interface:

- `bundle-name` – Symbolic name of a bundle providing the interface.
- `bundle-version` – Version of a bundle providing the interface.

Bundle installation or update must be canceled, if one of the following conditions is met:

- Directive or attribute is written twice in the definition.
- `Bundle-name` or `bundle-version` is specified in the providing interface definition.

Definition of the providing interface doesn't automatically mean that this interface is actually imported. A bundle that provides some interface and simultaneously doesn't import this interface can acquire the interface from its class path (B`undle-Classpath` definition), but not from other bundles.

Example of correct definition:

```
    Provide-Services: com.acme.Foo;com.acme.Bar;version=1.23
```

3.1.3.8 Cosi-Version

Version of CoSi specification for which was the component designed.

```
Cosi-Version ::= number
```

3.1.3.9 Bundle-Description

Short description of usage and purpose of the bundle.

3.1.3.10 Bundle-Provider

Name of the creator (developer, supplier) of the bundle.

3.1.3.11 Bundle-Classpath

`Bundle-Classpath` expression defines a list of *.jar files or directories inside the bundle file, where source files, compiled classes or other resources needed to run the bundle, will be searched for. Items of the list are separated by dash. Dot ('.') specifies root directory of bundle JAR. This dot is also default value for this `Bundle-Classpath` (root directory is always included into the class path).

Example:

```
     Bundle-Classpath: ., /jar/http.jar, /jar/log4j.jar
```

3.1.3.12 Generate-Events

Specifies events generated by the bundle. Programmer must specify event (message) name (message) and event type (class). Type is specified in the "type" mandatory header parameter. Type of the event isn't automatically exported by the container, bundle programmer must explicitly export this type in `Provide-Types` manifest header. Only event name and type is specified in `Generate-Event` header. More metadata of this type (version, etc.) are specified in `Provide-Types`.

Parameters that can be specified for event:

`version` – Isn't mandatory. Version of provided event. Default value is 0.

3.1.3.13 Consume-Events

Specifies events required by the bundle. Together with event name definition event type must be present in "type" parameter. Type is stated with its full name (class name including corresponding package name). Because type (class) of event is almost always some special type implemented by other bundle, the bundle must import that type in `Require-Types` manifest header. There is no additional information stated when defining imported events. This information (version, name of exporting bundle) is present in `Require-Types` manifest header.

Parameters that can be defined for required event:

`versionrange` – Optional. Specifies imported event version range. Default value is `[0.0.0, ∞)`.

3.1.3.14 Provide-Attributes

Specifies attributes provided by the bundle. Together with attribute name definition attribute type must be present in "type" parameter. Type is stated with its full name (class name including corresponding package name). Type of attribute must be exported or imported by the bundle in order to be processed correctly by the container. There is no additional information stated in attribute definition. This information is defined in `Provide-Types` manifest header.

Parameters that can be specified for attribute:

`version` – Optional. Version of provided attribute. Default value is 0.

3.1.3.15 Require-Attributes

Specifies attributes provided by the bundle. Together with attribute name definition attribute type must be present in "type" parameter. Type is stated with its full name (class name including corresponding package name). Type of attribute must be imported (in `Require-Types` header) by the bundle in order to be processed correctly by the container. There is no additional information stated in attribute definition. This information is defined in `Require-Types` manifest header.

Parameters that can be specified for imported attribute:

`versionrange` – Optional. Specifies imported attribute version range. Default value is `[0.0.0, ∞)`.

3.1.3.16 Provide-Packages

Specifies which class packages are exported by the bundle; cf. clause 3.5.5 of [OR4]. The semantics is that all public classes (as available through introspection) contained in the packages are made available to potential importers, unless enumerated explicitly by Provide-Types header.

Parameters that can be specified for imported attribute:

`version` – Optional. Specifies the version of the package as exported by the bundle.

3.1.3.17 Require-Packages

Specifies which class packages are imported by the bundle, cf. clause 3.5.4 of [OR4]. The semantics is that all public classes (as available through introspection) contained in the packages are required by the bundle, unless enumerated explicitly by Require-Types header.

Parameters that can be specified for imported attribute:

`resolution` – Optional, default “mandatory”. Value “optional” indicates bundle can resolve even when no provider of the package is found.

3.1.3.18 Provide-Types

Specifies which types are exported by the bundle; overrides Provide-Packages header.

Provided types definition entry has same syntax as provided interfaces definition entry. The difference is that types are classes exporting bundle wants to offer to other bundles but these classes cannot be interpreted as services. If a bundle imports a type, the type is added to bundle's class path so that bundle can use it.

3.1.3.19 Require-Types

Specifies which types are imported by the bundle; overrides Require-Packages header.

Required types definition entry has same syntax as required interfaces definition entry. The difference is that imported types are classes that bundle needs to run but these classes cannot be interpreted as other bundle's services. If there is a bundle providing imported type in the system, this type is added to importing bundle's class path.

3.1.3.20 Bundle-ExtraFunc

CoSi v2 enables specification of extra-functional properties on a bundle or on its feature. For *bundle-wide properties*, a new manifest header is added:

```
Bundle-ExtraFunc: <name>=<value>, ...
```

For *feature-related properties* the OSGi attribute syntax is used:

```
<manifest-header>: <object>; extrafunc=(<name>=<value>, ...)
```

The name part is a comma-separated identifier, where commas serve to distinguish “scope” of the property. The value part uses format-based denotation of value type chosen so that property type can be inferred from the value by lexical scanner, without further hints to the processor. This may change in future revisions of the specification as more enhanced properties arise.

Property types

The following table lists the types recognized.

Type (scalar)	*Format of values*	*Note*	*Examples of values*
boolean	true|Y, false|N		Y
integer	32-bit signed int	decimal point required	-30220
float	64-bit IEEE format float	decimal point required	0.25 34.0
token	identifier with dashes and	As per 1.4.2 of [OR4]	embedded first-line

	underscores		
string	double-quote delimited string		“That's enough, mate”
Type (complex)	*Format of values*	*Note*	*Examples of values*
enum<*type*>	{ value [, value, ...] }	Values MUST be of the same scalar *type.*	{ start, stop, unknown }
map	{ identifier: value, ... }	Values NEED NOT be of the same scalar *type.*	{ memory: low, security: C2 }
interval<*type*>	(val_{min}, [val_{mean},] val_{max}]	Values MUST be of the same scalar *type.* Uses OSGi interval notation.	(0.1, 1.0] [xsmall, large, xxlarge]

Property functions

The following table lists functions available for manipulating / deriving properties. It is expected that they will be used primarily on the headers declaring required features.

Function	*Signature*
comparison	< , <=, >, >=, =, <>
min	min(<complex-type>) → value Returns or denotes first enumeration element or lowest value of an interval.
max	max(<complex-type>) → value Returns or denotes last enumeration element or highest value of an interval.
median	median(<complex-type>) → value Returns or denotes the value in the middle of an enumeration or interval.
avg	avg(<complex-type>) → value Returns or denotes the arithmetic average value of an enumeration or interval (with integer or real values).
contains	contains(<complex-type>, {<value>[, <value>, ...]}) → bool Queries whether the type contains the given values.

Example:

```
Bundle-ExtraFunc: memory=32, security={basic,digest},
      gc-model=”eager v3”, precision={normal:”0.01”, high:”0.0001”}
```

```
Export-Types: cz.zcu.kiv.cosi.weather.tempsensor.Sensor;
```

```
      extrafunc=(precision=normal, perf.refresh=500),
      cz.zcu.kiv.cosi.weather.tempsensor.ControlPoint
Import-Types: com.acme.actuator.TempActuator; extrafunc=(refresh<=400),
      com.other.comm.SecureDataTransferer; version="[1.0,
      2.3)";extrafunc=(contains(security.method,{ssl,tls}),
      avg(perf.delivery_time)<=10)
```

This property header and attribute assume the existence of a property definition registry referenced by the ExtraFunc-Catalog header and described in section 3.1.4 Extra-functional property registry and query service below.

3.1.3.21 ExtraFunc-Catalog

This header specifies the URI location of the extra-functional property registry (see section 3.1.4). If the header is missing in the manifest file, extra-functional property types are considered undefined and the `Bundle-ExtraFunc` header as well as the `extrafunc` attribute values are discarded.

Example:

```
      ExtraFunc-Catalog: http://services.kiv.zcu.cz/cosi/extrafunc/v1/
```

3.1.3.22 Other rules concerning the manifest.mf file

- Each definition can be present more than once in manifest.mf file, but only the last one stated is valid.
- Manifest can contain additional definitions that are not stated in this document. These definitions are loaded, but only as a string without meaning that is not interpreted by the framework. These values are accessible, but have no impact on bundle installing.

3.1.4 Extra-functional property registry and query service

To enable reasoning about extra-functional properties, CoSi is enhanced with a simple registry that contains complete extra-functional property type definitions. In the current CoSi version, it uses a simple text syntax of registry file with the following format:

```
      # hashmark starts a comment line
      <prop-def> := <name> : <type> [; <constraint>]
      <type> ::= see the names of types in section
      <constraint> ::= <interval-constr> | <set-constr> | <map-constr>
      <interval-constr> ::= "<" <value> , <value> ">"
      <set-constr> ::= "{" <value [, ...] "}"
      <map-constr> ::= "{" <name> : <value> [, ...] "}"
```

The registry is accessible in a CoSi container via a standard service – see section 3.3.2 ExtrafuncRegistry Service.

3.1.5 Bundle dependencies

This section details the way bundle dependencies are implemented.

3.1.5.1 Service acquiring and registering

Bundle can acquire or register only service instances stated in the manifest file – in `Require-Interfaces` or `Provide-Interfaces` header. If a bundle tries to register a service that is not stated in `Provide-Interfaces` header in the manifest file, following steps are taken:

1. The frameworks logs that attempt and prints an error on the console
2. The service won't be registered
3. Bundle runtime won't be affected – the framework won't throw an exception

Properties describing the service can be attached to each service that bundle wants to register. It is set of pairs `key-value`, both `String` type. In the following text, these properties are named *service properties.*

These properties are desirable in case when programmer expects that there will be more service instances present in the system. The container allows to register unlimited number of services for one interface. For that reason, it may be difficult for other bundles to pick the right service instance. Information available in service properties help bundles to choose particular service instance. Service properties are dynamic – property keys that will be attached to the services are not stated in the manifest file. It is absolutely up to the programmer, what information will he state in service properties.

When acquiring service instances, it is possible to specify similar set of properties. The container takes care of returning a service instance with corresponding properties. Each required properties entry is compared with registered services properties and when match is found, the container returns found service.

When we call properties specified during service registering “registered properties” and properties specified during service instance acquisition “required properties”, we can describe equality algorithm as following:

1. If there is a specific key in required properties, this key must be also in registered properties. Corresponding values must be equal.
2. If there is a key in registered properties that is not present in required properties, the key is ignored.

In other words required properties must be subset of registered properties.

Bundle programmer can also use methods that completely ignore properties, more accurately register and acquire services with empty set of properties. Because of that, in case of service instances acquisition, it isn't specified, what instance will be returned to the bundle. Methods `registerService(String, Object)` and `getService(String)` are used in this case.

Last way how to treat information about a service when acquiring service instance is `getServiceReferences(String, HashMap<String, String>)` method. This method returns list of all services complying with a given type (first argument), and whose registered properties comply with requested properties (second argument). This method returns *reference* to all matching services. These references (`ServiceReference`) can be used to acquire additional information about a service or to get service instance. Bundle is allowed to acquire reference to any service in the system. Service doesn't have to be in bundle's manifest file in order to acquire the reference.

3.1.5.2 Attribute usage

All attributes used by a bundle during runtime must be stated in the manifest file. Rules for using attributes are a little bit more complicated compared to services.

Bundle is allowed only to read attribute stated in `Require-Attributes` header in the manifest file. If an attribute has some special type not provided by the container, attribute type must be stated in `Require-Types` part of the manifest file. That means bundle must import corresponding type.

If an attribute is stated in `Provide-Attributes` header in the manifest file (bundle provides the attribute), bundle is allowed to read and write this attribute. If the attribute has a special type, this type must be stated in `Provide-Types` or `Require-Types` header.

3.1.6 Bundle distribution form

A bundle must be distributed as one *.jar (JavaArchive) file, which contains all necessary things needed to run a bundle. The framework encloses bundle to its *.jar file and forbids using resources from outside the *.jar file.

Bundle archive must contain manifest.mf file inside the META-INF directory, with specification information about the bundle.

The archive must include additional mandatory directories as stated below. Container checks for the presence of these directories during component installing and if they are not present, component installing ends with an error and component is not installed into the container. Optional directories are mentioned too in order to establish directory naming convention.

Directories within the *.jar file (optional directories are distinguished by the lighter color):

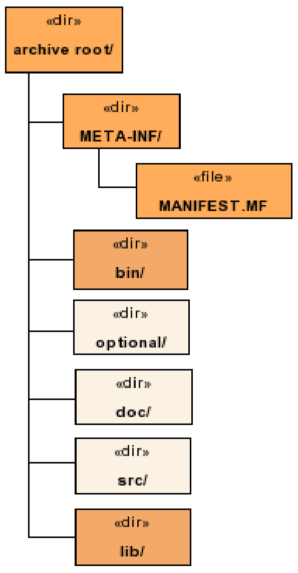

Fig.1: Bundle distribution form

- **bin/** directory – Mandatory directory containing compiled component code. This directory is in the bundle classpath by default.
- **lib/** directory – Mandatory directory containing libraries needed to run the component. These libraries can be stores as *.jar files. This directory can be empty if bundle doesn't need any libraries, but directory itself must be present. Libraries used by the bundle must be stated in classpath in the manifest file.
- **imports/** directory – Mandatory, contains class files (or .jar files containing them) of all types in the imported packages/types. One level of referenced imported types is mandatory, transitive closure is recommended.
- **optional/** directory – Optional, can contain miscellaneous files and directories.
- **src/** directory – Optional, contains source code of the bundle, if bundle maintainer wants to publish them.
- **doc/** directory – Optional directory containing bundle documentation.

3.2 Bundle Lifecycle

CoSi bundle lifecycle is an extended version of OSGi bundle lifecycle. A set of bundle states is recognized which can be queried via the `Bundle.getState()` method, and a set of events is fired by the container upon state changes.

The following states are recognized (see also Figure 2: CoSi bundle lifecycle):

- INSTALLED
- RESOLVED (synonym: STOPPED)
- STARTING
- STARTED (synonym: RUNNING)
- STOPPING
- UPDATING
- UNINSTALLED

For details of the resolution step see section 3.2.2 Resolving. A bundle is said to be *active* if it is in one of the states STARTING, STARTED, STOPPING.

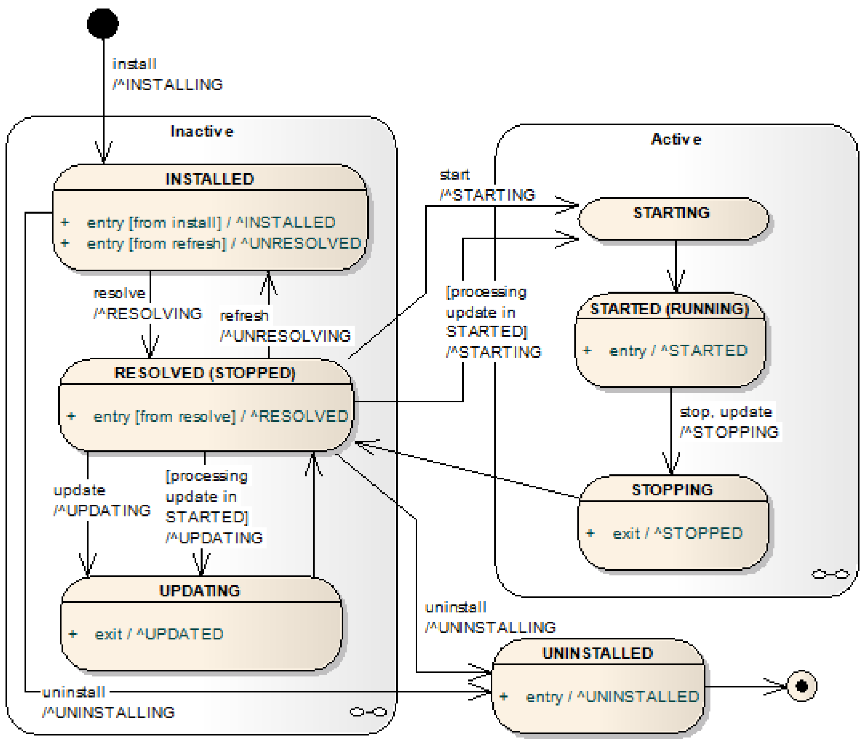

Figure 2: CoSi bundle lifecycle

3.2.1 Lifecycle Events

The following events are fired on state transitions (see also Figure 2: CoSi bundle lifecycle):

- INSTALLING

- INSTALLED
- RESOLVING
- RESOLVED
- UNRESOLVING
- UNRESOLVED
- STARTING
- STARTED
- STOPPING
- STOPPED
- UPDATING
- UPDATED
- UNINSTALLING
- UNINSTALED

The events correspond to commencing the transition (start of the corresponding activity) and entering the state (finish of the activity). Together with the standard message consumers, or event listeners, they thus allow intercepting and possibly influencing the state transitions by other bundles.

The events are represented by instances of the `cz.zcu.kiv.cosi.systemservice.Bundle-LifecycleMessage` class which extends the `cz.zcu.fav.kiv.cosi.messageservice.-Message` class. It has the following properties:

- The `getMessageName()` method always returns "COSI.BUNDLE_LIFECYCLE_MESSAGE".
- It implements a `public String getEventName()` method which returns a string with event name as defined above.

The framework always sends the events via the standard message service (see section 3.3.2) using synchronous delivery (the `sendMessage()` method).

3.2.2 Resolving

First step of bundle installation is *bundle resolving*. It is a process, in which providers are being searched for for particular bundle imports (type imports, attribute imports, message imports). Bundles can have constraints for their imports such as specific interface version, exact name of providing bundle, etc. If no bundle matching criteria is found, the bundle is *not* correctly resolved (it means bundle can't be started). Bundle stays in the system in INSTALLED state, so it's possible to attempt to resolve the bundle later, under other conditions. Bundle resolving must be done before any bundle code is loaded or ran.

During the resolving, container checks:

1. if bundle is in INSTALLED or RESOLVED state. Bundle must be correctly installed in order to be able to be resolved. It is possible to resolve already resolved bundle.
2. if there are providers for all imported interfaces. Only presence of providers is checked during bundle resolving, no connections are made between bundles.

3. if there are providers for all imported types. Only presence of providers is checked during bundle resolving, no connections are made between bundles.
4. if there are providers for all imported attributes. Only presence of providers is checked during bundle resolving, no connections are made between bundles.
5. if there are providers for all imported messages. Only presence of providers is checked during bundle resolving, no connections are made between bundles.
6. if provided attribute types are correctly exported (if type class is provided). This rule can be broken if bundle imports particular attribute type. It is not necessary to export types included in *Boot class path*.
7. if provided messages types are correctly exported. Same rules are applied as for attribute types.
8. if bundle was in RESOLVED state before resolving, resolving process ends with this step. If all precedent steps were successful, bundle is left in RESOLVED state. If at least one of the steps failed, bundle is moved into INSTALLED state and all following steps must be negated – container must return to the state it was before first (successful) resolving of this bundle. If bundle was in INSTALLED state, resolving process continues normally with the next steps.
9. All resources bundle provides are registered with the container (interfaces, types, attributes or messages).
10. Bundle is linked with its providers. There is no *wire* object for this like in OSGi, so linking can be implemented with simple list of exports and imports.
11. Instance of bundle activator is created.

If all steps are successfully completed, bundle is moved into the RESOLVED state and it's ready to be started.

When checking interface, attribute, types or messages providers, constraints must be taken into the consideration to make sure bundle is linked with the correct provider.

3.2.3 Bundle interfaces

The CoSi framework defines several interfaces which are directly related to the bundle instance and its features. The interfaces are

- Bundle – represents the bundle, provides lifecycle methods; described below in detail.
- BundleMetaData – provides access to information in bundle manifest.
- ServiceReference – represents a provided service.
- ModuleClassLoader – the contract of classloader used within CoSi container.

The interfaces not described here are defined by their source code documentation.

3.2.3.1 Bundle interface

Each bundle installed in a container is represented by a singleton object of this interface, created by the container. For detailed description of methods see source documentation.

Interface definition:

```
public interface Bundle {

      public static final int INSTALLED = 0;
      public static final int RESOLVED = 1;
      /* ... etc for each of the states defined in section 3.2 */

      public String getBundleLocation();
      public boolean installBundle();
      public boolean uninstallBundle();
      public boolean resolveBundle();
      public void startBundle();
      public boolean stopBundle();
      public void updateBundle(URL Ps_newBundleUrl);

      public boolean isInstalled();
      public boolean isResolved();
      public boolean isStarted();
      public boolean isActive();
      public int getState();

      public BundleMetadata getBundleMetadata();

      public int getId();
      public String getName();
}
```

3.2.3.2 BundleMetaData interface

Provides access to meta-data items from the manifest file. For referenced classes see the source code documentation.

Interface definition:

```
public class BundleMetadata {

      public boolean isManifestValid();

      public String getCosiVersion();
      public String getBundleName();
      public Version getBundleVersion();
      public String getBundleDescription();
      public String[] getBundleClasspath();
      public String getControlClass();
      public String getBundleProvider();
      public ExtraFunc getBundleExtraFunc();

      public List<ProvidingTypeHeaderEntry> getProvideServices();
      public List<RequiringTypeHeaderEntry> getRequireServices();
      /* @deprecated */
      public List<ProvidingTypeHeaderEntry> getProvideInterfaces();
      /* @deprecated */
      public List<RequiringTypeHeaderEntry> getRequireInterfaces();
      public List<ProvidingNamedTypeHeaderEntry> getGenerateEvents();
      public List<RequiringNamedTypeHeaderEntry> getConsumeEvents();
      public List<ProvidingNamedTypeHeaderEntry> getProvideAttributes();
      public List<RequiringNamedTypeHeaderEntry> getRequireAttributes();
      public List<ProvidingTypeHeaderEntry> getProvideTypes();
      public List<RequiringTypeHeaderEntry> getRequireTypes();
      public List<ProvidingTypeHeaderEntry> getProvidePackages();
      public List<RequiringTypeHeaderEntry> getRequirePackages();
}
```

3.3 The Container

The container is part of the CoSi core responsible for:

- bundle loading, bundle life cycle maintaining
- keeping register of services provided by bundles
- keeping list of attributes provided by bundles
- providing information about bundles – their state, dependencies, metadata, etc.
- providing services for controlling bundle life cycle
- providing service a bundle can use to send or receive messages

The container is a service on background that holds no functionality itself, but provides *environment* for bundle runtime like operating system provides environment and runtime for applications.

If bundle A provides types, attributes or messages and there is bundle B in the container using (importing) one of these elements, bundle B must be stopped and uninstalled in order to be able to uninstall bundle A. This design limitation provides correct runtime for bundles. Unlike is OSGi, there are no message generators announcing uninstalled services to other bundles. While the bundle is running, it is ensured that all services acquired by the bundle during runtime are present in the system until `stop()`method is called.

3.3.1 Container interfaces

Even thought bundle is considered a black box – environment has no information about bundle except provided and required items (interfaces, attributes, etc.), it is necessary for a bundle to be able to communicate with the framework and for the framework to be able to communicate with the bundle. The framework provides two interfaces to ensure this functionality. First interface is `BundleControl` that must be implemented by every bundle and stated in `Control-Class` manifest header. The framework controls bundle life cycle through this interface.

Second interface provided by the framework is `BundleContext` interface. This interface provides basic information about bundle environment.

3.3.1.1 BundleControl interface

`BundleControl` is an interface that *must* be implemented by one bundle class so that the container could start or stop it. The container creates instance of the class stated in `Control-Class` header in the manifest during resolving and checks if the class correctly implements this interface. If `start()` method is successfully called from this instance, it is ensured that during bundle stopping, `stop()` method will be called on the same `BundleControl` instance.

In order to be able to create instance with `Class.newInstance()`, class implementing `BundleControl` must have public constructor without parameters.

Right before calling `start()` method, bundle is moved into the STARTING state and remains in this state until successful completion of this method. If there are no problems during `start()` method calling, bundle is moved into the STARTED state a is considered running. By analogy, right before `stop()` method calling, bundle is moved into the STOPPING state and remains in this state until successful completion of this method. In this case bundle state returns to RESOLVED.

Container ensures safe initiation of both operations. Any error or exception that occurs during starting or stopping bundle must be caught so that other bundles weren't affected by this error (exception).

Rules for error calls of operations:

For `start()` method:

- Container must log exception generated by the start() method.
- Container must return to the state before the start() method was called. If bundle managed to register some services or attributes before exception occurred, these services (attributes) must be removed by the container.
- Bundle returns to the RESOLVED state.

For `stop()` method:

- Container must log exception generated by the start() method.
- It is very likely that bundle didn't manage to unregister all services or attributes registered during its runtime. For that reason it is necessary for a container to remove all services or attributes registered by the bundle.
- Bundle returns to the RESOLVED state.

`Start()` method is the place where bundle should acquire links to other services or register own services or attributes. Service (attribute) acquisition or registration is done using `BundleContext` interface provided as a parameter. The framework ensures that method will be called only *once* during bundle startup and while bundle is active, any attempt to call the method will be blocked (you cannot start already started bundle). If the bundle is stopped before calling `start()` method (using `stop()` method), it *is possible* to start it again.

`Stop()` method is the place where bundle *must* unregister all its services, attributes and messages. `Stop()` method is called with the same `BundleContext` instance as `start()` method.

`BundleControl` interface definition:

```
public interface BundleControl {
      public void start(BundleContext P_bd);
      public void stop(BundleContext P_bd);
}
```

3.3.1.2 BundleContext interface

Bundle context object is used for communication with the framework, framework access and for access to services registered inside the framework. Bundle can register its services or acquire references to other services using this object. Apart from these functions, `BundleContext` provides the following functionality for the bundle:

- Service registration and acquisition
- Bundle metadata acquisition
- Using standard input and output provided by the framework

- Write or read attributes

`BundleContext` instance is created during bundle start and is passed to the bundle through `BundleControl.start(BundleContext)` method. The same instance of this class is passed to the bundle associated with this context during `BundleControl.stop(BundleContext)` call. This context is considered close part of the bundle and should be used with associated bundle. Sharing context is considered inappropriate in the CoSi framework.

Context instance is valid only during active state of the bundle – only in STARTING, STARTED or STOPPING state. Context instance can't be used if bundle isn't in on of these states. When reusing the bundle, new context instance is created.

Framework is the only entity that is allowed to create `BundleContext` instance and these instances are valid only inside the framework that created them.

`BundleContext` interface is defined as:

```
public class BundleContext {
      public void registerService(String clazz, Object service);
      public void registerService ( String clazz,
            Object service, HashMap<String, String> P_properties);
      public void unregisterService(Object service);
      public Object getService(String clazz);
      public Object getService(String Ps_clazz,
                              HashMap<String, String> P_properties);
      public Object getService(ServiceReference P_reference);
      public ServiceReference[] getServiceReferences(String clazz,
            HashMap<String, String> P_properties);
      public InputStream getStdIn();
      public OutputStream getStdOut();
      public void setAttributeValue(String Ps_attributeName, Object
P_attributeValue);
      public Object getAttributeValue(String Ps_attributeName);
      public BundleMetadata getBundleMetadata();
}
```

Method description:

`public void registerService(String Ps_clazz, Object P_service);`

Registers service instance `P_service` with the container. Service is listed as `Ps_class` in the container, which is fully qualified name of the service. Type of the `P_service` argument must match with this name. During service registration, ServiceReference type object is created and attached to registered service instance. This object is accessible by calling `getServiceReferences` method. During registration is also being checked, if bundle is allowed to register particular service type (if that type is stated in `Provide-Interfaces` header). Service is registered with *empty* properties.

```
public void registerService ( String clazz,
            Object service, HashMap<String, String> P_properties);
```

This method has same usage and behavior as `registerService(String,Object)`, but we can specify service properties. `P_properties` attribute can't be null.

`public void unregisterService(Object service);`

Unregisters previously registered service.

`public Object getService(String Ps_clazz);`

This call of getService without properties specified means that programmer doesn't care about properties, and just want some service which implements given class. Checks must be made that bundle has right to obtain service with this class - this means that bundle has this `Ps_clazz` mentioned in it's manifest in `Require-Interfaces` part. If it is not there a null must be returned.

```
public Object getService(String Ps_clazz,
                    HashMap<String, String> P_properties);
```

Returns *first* service which conforms to the properties given. If no such service exist a null value is returned. Checks must be made that bundle has right to obtain service with this class - this means that bundle has this `Ps_clazz` mentioned in it's manifest in `Require-Interfaces` part. If it is not there a null must be returned.

```
public Object getService(ServiceReference P_reference);
```

Returns service associated with a `ServiceReference` object. Checks must be made that bundle has right to obtain service with this class - this means that bundle has this `Ps_clazz` mentioned in it's manifest in `Require-Interfaces` part. If it is not there a null must be returned.

```
public ServiceReference[] getServiceReferences(String clazz,
            HashMap<String, String> P_properties);
```

Returns service references to all services which conforms to a given properties filter. Bundle is allowed to obtain `ServiceReferences` even to those services, which are not specified in it's manifest.

```
public InputStream getStdIn();
```

Returns standard Framework input stream.

```
public OutputStream getStdOut();
```

Returns standard Framework output stream.

```
public void setAttributeValue(String Ps_attributeName, Object
      P_attributeValue);
```

Sets the framework-scope attribute (property). If a bundle wants to operate on some attribute, this attribute has to be specified in `Provide-Attribute` or `Require-Attribute` part of the manifest file. In the manifest file is also specified type of the attribute and this type has to be used when setting attribute.

```
public Object getAttributeValue(String Ps_attributeName);
```

Retrieves value of the framework-scope attribute (property). If a bundle wants to operate on some attribute, this attribute has to be specified in Provide-Attribute or Require-Attribute part of the manifest file. In the manifest file is also specified type of the attribute - this is the type of returned value.

```
public BundleMetadata getBundleMetadata();
```

Returns context bundle metadata (parsed manifest file).

3.3.2 System services

Container provides basic API for bundles in the form of two services:

- `SystemService` is a system service used by bundles to control other bundle's life cycle and to obtain information about container inner state.

- MessageService is a system service used by bundles to send and receive messages.
- ExtraFuncService is a system service providing extra-functional properties for bundles.

All three services are *always* available to all bundles. That means container has to start these services immediately after its start and before any other bundle installation or start. Special pseudo-bundle is created for each service whose only task is to provide the service. Pseudo-bundle word means that implementation doesn't have to result from classic bundle, however there will be “systembundle” bundle providing “SystemBundle” service etc., visible to other bundles in the system. Other bundles can treat these bundles in the same way as they treat other bundles with the exception of stopping the bundle.

SystemService is installed and started as a first bundle, MessageService as a second. ExtraFuncService is started as a third.

3.3.2.1 System Service

Framework must contain an implementation that registers SystemService service after its start. This service provides basic access into the system for bundles. Interface of this service is part of the framework and is located in cz.zcu.fav.kiv.cosi.systemservice package. Other bundles can install other bundles or perform operations related to life cycle through the system service. Service also contains method providing information about bundle dependencies. The system service bundle exports the LifecycleEventMessage mentioned above so that user bundles can declare appropriate dependency to intercept bundle lifecycle events.

Instance of this service is always present as a service with id 0 and started before any other bundles are started or installed. Bundles don't have right to stop system services. Implementation of this service has basically only small functionality and is meant to be a delegate to request framework core functions.

SystemService interface definition:

```
public interface SystemService {
   public HashMap<Integer, Bundle> getBundles();
   public Bundle getBundle(int Pi_id);
   public Version getContainerVersion();
   public void shutdownContainer();

   // bundle life cycle
   public void installBundle(String Ps_bundleId);
   public void uninstallBundle(int Pi_id);
   public void startBundle(int Pi_id);
   public void stopBundle(int Pi_id);
   public void updateBundle(int Pi_id, String Ps_newBundleURL);

   // information about bundle state
   public Collection<ProvidingTypeHeaderEntry> getExportedTypesForBundle(int
Pi_id);
   public Collection<RequiringTypeHeaderEntry> getImportedTypesForBundle(int
Pi_id);
   public Collection<ProvidingTypeHeaderEntry>
getExportedInterfacesForBundle(int Pi_id);
   public Collection<RequiringTypeHeaderEntry>
getImportedInterfacesForBundle(int Pi_id);
   public Collection<NamedProvidingTypeHeaderEntry>
getProvidedAttributesForBundle(int Pi_id);
   public Collection<NamedRequiringTypeHeaderEntry>
getConsumedAttributesForBundle(int Pi_id);
```

```
   public Collection<NamedProvidingTypeHeaderEntry>
getProvidedMessagesForBundle(int Pi_id);
   public Collection<NamedRequiringTypeHeaderEntry>
getConsumedMessagesForBundle(int Pi_id);
   public Collection<ServiceReference> getInstalledServicesForBundle(int Pi_id);
}
```

3.3.2.2 Message Service

The framework must contain a bundle that registers the `MessageService` service during its start. The bundle will obtain id 1 during startup. This service allows other bundles to send and receive messages (events). Bundles use `Message`, `MesageService` and `MessageConsumer` types to create, send and receive messages.

If a bundle wants to generate new event, it implements abstract class `Message` representing abstraction of event used in CoSi environment. This created object is send to the service using `sendMessage()` or `postMessgage()` methods. The message is delivered to all bundles capable to receive the message.

Bundle must implement `MessageConsumer` interface in one of its classes in order to be able to receive messages. This object is then registered with `MessageService` service using `registerMessageConsumer()` method. The service delivers messages to this object to be processed.

Service controls if bundle sending or receiving a message is allowed to do that action. Information stated in the manifest file must correspond with message contents. Service grants right to send only messages stated in `Generate-Events` header and to receive only messages stated in `Consume-Events` header in the manifest file.

Before the message is delivered, service implementation must check that sending bundle is still in active state. If so, the message is delivered normally. However, if the bundle isn't active, all `MessageConsumer` interface implementations registered by the bundle must be removed from the service and no messages can be delivered to these *receivers*. This condition is present because of possible fail during bundle termination. If an exception occurs during bundle termination and because of that, all receivers aren't correctly unregistered, `MessageService` service must unregister these receivers by checking bundle active state.

Instance of this service is *always* present in the container with id 1 and is started right after `SystemService` service and before any other bundle. Other bundles, except `SystemService`, can't stop `MessageService`.

`MessageBundle` interface definition:
```
public interface MessageService {
      public void registerMessageConsumer(MessageConsumer P_consumer,
                                Bundle P_consumingBunde);
      public void unregisterMessageConsumer(MessageConsumer P_consumer);
      public void sendMessage(Message P_message);
      public void postMessage(Message P_message);
}
```

3.3.2.3 ExtrafuncRegistry Service

The framework must contain a bundle that registers the `ExtrafuncRegistryService` service during its start. The bundle will obtain id 2 during startup.

This service provides the facility to check bundle's extra-functional properties against the property registry and verify if one bundle extra-functional properties provide all required extra-functional properties of another bundle.

`ExtraFuncRegistryService` interface definition:

```
public interface ExtrafuncRegistryService {
      /** Evaluate substitutability of two properties. */
      public boolean checkExtraFuncMatch(ExtraFunc lhsExtraFunc,
            ExtraFunc rhsExtraFunc);
      /** Verify property correctness. */
      public boolean checkExtraFuncValid(ExtraFunc extrafunc);
      /** Initialize registry from a definition file. */
      public void loadRegistryFromURL(URL url) throws Exception;
}
```

3.3.3 Additional services description

In addition to the core system services, the default CoSi framework distribution may contain the following standardized services, to be used by user bundles.

3.3.3.1 SimpleShell service

One of the basic services for user interaction is a command line implemented by `simpleshell.jar` bundle. Using this command line, user can enter commands and control the container. Standard commands include bundle life cycle control (installation, starting, stopping, updating, etc.) and getting information about the container state and system state. Except this, `SimpleShell` provides service that can be used to register new commands. The command interface is used for this purpose. Bundles can use this service to create simple user interface.

The simple shell bundle is written in Groovy.

Command interface definition:

```
public interface Command {
      public String getHelp();
      public void doCommand(String[] P_arguments);
      public String getName();
}
```

To use this service, bundle must specify following headers in its manifest:

```
      Require-Interfaces:  cz.zcu.fav.simpleshell.SimpleShell

      Require-Types: cz.zcu.fav.simpleshell.Command
```

3.3.3.2 LogService service

Bundle `logservice.jar` provides a service for uniform bundle logging. Every bundle could solve it with own logger, yet this service provides central log administration. Bundle is based on `log4j` library.

Definition of the provided LogService interface:

```
public interface LogService {
      public static final int LOG_DEBUG = 10;
      public static final int LOG_INFO = 20;
      public static final int LOG_WARN = 30;
      public static final int LOG_ERROR = 40;

      public void configure(InputStream P_log4jConfigFile);
      public void setGlobalLevel(int Pi_level);
      public void setLevelFor(String Ps_ident, int Pi_level);
      public void log(int Pi_level, String Ps_message);
      public void log(int Pi_level, String Ps_message, Throwable P_exception);
      public void log(String Ps_ident, int Pi_level, String Ps_message);
      public void log(String Ps_ident, int Pi_level, String Ps_message,
Throwable P_exception);
}
```

The service provides a system based on loggers that can be set to certain level. These levels are defined as constants in service interface (`cz.zcu.fav.kiv.logservice.LogService`). If a logger is used to log a message, one of these levels is assigned to the message. Accordingly as logger level is set, it is decided, whether message is or isn't logged. If message level is higher or same as logger level, message is logged and vice versa.

Service provides one pre-defined logger that can be used by bundles if they don't care where will the messages be written. It is possible for a bundle to define own logger.

3.3.3.3 HttpService service

`HttpService` service implemented by `httpservice.jar` bundle provides simple web server based on *servlets* similar to those in J2EE specification, but simplified. Server implementation is very plain. Only basic HTTP requests are supported (GET and POST). Bundle comes with a service that other bundles can use to register own servlets with the web server.

Definition of the provided HttpService interface:

```
public interface HttpService {
      public void registerServlet(String Ps_mapping, Servlet P_servlet);
      public void unregisterServlet(Servlet P_servlet);
}
```

3.4 Class loading architecture

CoSi class loading architecture is similar to OSGi class loading architecture, but it's simplified (see [OR4]). All the bundles installed in one container share a single virtual machine (VM). Within this VM, bundles can hide packages and classes from other bundles, as well as share packages with other bundles. The key mechanism to hide and share packages is the Java class loader that loads classes from a sub-set of the bundle-space using well-defined rules. The special class loader adhering to the rules listed hereinafter must be used in CoSi framework.

Every bundle has a single class loader that takes care of bundle resource loading. That class loader forms a class loading delegation network with other bundles.

The class loader can load classes and resources from:

Boot class path – The boot class path contains the `java.*` packages, basic Groovy packages and its implementation packages. Only the framework can provide these basic resources. A bundle can expect that these resources are available and is not allowed to add them to its class path.

Framework class path – The framework provides basic set of interfaces and types needed to create bundles (for ex. `BundleContext`, `BundleControl`). These types and interfaces are automatically added to the bundle class path.

Bundle class path – The bundle can define its own class path. This class path is defined in the manifest file and is relevant only for *the inside* of the bundle. It can refer only to directories or archives inside the bundle archive.

A class space is then all classes reachable from a given bundle’s class loader. Thus, a class space for a given bundle can contain classes from:

- The parent class loader (normally classes from boot class path and framework class path)
- Imported packages
- Imported types
- The bundle's class path

A class space must be consistent, such that it never contains two classes with the same fully qualified name (to prevent Class Cast Exceptions). However, separate class spaces in an CoSi may contain classes with the same fully qualified name.

3.5 Differences from CoSi v1

The version of the CoSi component model and framework specified in this document (version 2) differs from the first version [Waj07] is the following major aspects. (This is a summary of the changes, for detailed information please see the individual sections.)

New features introduced

- Support of extra-functional properties on bundles and surface features.
- Container sends events on bundle state changes.
- Ability to specify bundle export and import
- Bundle package includes the imports/ directory with bundled imported classes.

Minor changes and corrections

- The *Provided-Services* and *Required-Services* headers are preferred over previously used **-Interfaces* headers.
- Component lifecycle has been enhanced.
- Internal improvements in the implementation

3.6 Differences from OSGi

Even though CoSi specification strongly results from OSGi framework specification, there are (often very fundamental) differences. This chapter describes these differences. Instead of detail description, how is particular thing handled on OSGi, only rough *specification* will be described in this chapter. Detailed OSGi specification can be found in [OR4].

Type Export/Import

The most important difference between both specifications is how bundles share their resources (what bundles export and import).

- Handled on package level in OSGi. Manifest file specifies what *package* from inside the bundle is provided or what *package* is required by the bundle.
- Handled on class level in CoSi. Manifest file specifies name of particular class that is exported or imported and this export/import can be versioned (see `Provide-Types`, `Provide-Interfaces`, `Require-Types`, `Require-Interfaces`). This design decision provided better control over bundle exports and imports. One drawback is when a bundle exports fifty classes from one package, they all have to be stated in the manifest file. In OSGi you only need one line to accomplish that.

Security

There is no security at any level specified in CoSi. In OSGi, there is optional mechanism of bundle validation. Certificates, keys, etc. can be used to verify components in OSGi. It was decided not to support such functionality in CoSi, because it isn't interesting functionality in bundle substitutability research.

Simplified class loading architecture

Working with class loading is a lot more complex in OSGi. Main reason are imports/exports on package level or for example so called *fragments* - incomplete bundles or possibility of dynamic imports.

Groovy

CoSi brings whole new possibilities of writing bundles in dynamically typed, fully interpreted language Groovy. CoSi allows class loaders to load source files in Groovy and treat them as normal Java files. In addition, such class loader must be able to load bundles written in Java only.

Native libraries

CoSi framework doesn't support working with native libraries (.dll, .so), while OSGi does.

Execution environment

OSGi specifies execution environment configuration as one of the manifest entries. This configuration must be fulfilled in order to install a bundle. It specifies for example required Java version or minimal OSGi version, etc. CoSi has no such configuration.

Manifest file and bundle structure

Manifest file and bundle structure differ in both frameworks. While directory structure is a lot alike, individual entries are completely different. Aside from header names being different, their meanings are different too.

Container state persistence

OSGi framework forces container implementations to support persistence. It means when framework is stopping, it has to stop and uninstall all bundles, but if one of the bundles was active, container must remember that and start and install all previously active bundles upon next framework start. This functionality is not required by CoSi. CoSi uninstalls all bundles when stopping, and the next start is “clean”.

Obligation to implement BundleControl

In CoSi framework bundles must implement the bundle activator – `BundleControl` interface. There is no such obligation in OSGi.

Enclosed JARs

In order to make bundle distribution easier, container provides possibility to read resources from jars from inside distribution jar (standard Java implementation doesn't allow that). However, this jar nesting can be used only for one level. It means jar file inside the bundle can't contain jar archives.

A Manifest File Syntax

This appendix defines the syntactical and other rules governing the manifest file entries.

A.1 Mandatory headers

These are the minimum that needs to be specified for any bundle. If any is missing, the bundle will not be installed.

- `Bundle-Name`
- `Bundle-Version`
- `Control-Class`
- `Bundle-Provider`

A.2 Semi-mandatory headers

The following headers must be present in the manifest file if the bundle wants to have the associated features or properties recognized by the container.

- `Provide-Interfaces`
- `Require-Interfaces`
- `Generate-Events`
- `Consume-Events`
- `Provide-Attributes`
- `Require-Attributes`
- `Provide-Types`
- `Require-Types`
- `Bundle-Extrafunc`

A.3 Optional headers

These headers are fully optional, that is no “harm” is done when if any of them is missing in the manifest file. All have reasonable default values which the container uses in that case.

- `Cosi-Version`
- `Bundle-Description`
- `Bundle-Classpath`

A.4 General Syntax Definitions

```
( )?     Optional part
*        Repetition of the previous element zero or more times,
              e.g. ( ',' element ) *
+        Repetition one or more times
?        Previous element is optional
( ... )  Grouping
'...'    Literal
|        Or
[...]    Set (one of)
..       list, e.g. 1..5 is the list 1 2 3 4 5
```

The following tokens are predefined and used throughout specifications:

```
digit          ::= [0..9]
alpha          ::= [a..zA..Z]
alphanum       ::= alpha | digit
token          ::= ( alphanum | '_' | '-' )*
number         ::= digit+
jletter        ::= <see Lexical Structure of Java Language for
                     JavaLetter>
jletterordigit ::= <see Lexical Structure of Java Language for
                     JavaLetterOrDigit >
qname          ::= <see Lexical Structure of Java Language for
                     fully qualified class names>
identifier     ::= jletter jletterordigit *
quoted-string  ::= '"' ( [^"\#x0D#x0A#x00] | '\"'|'\\')* '"'
argument       ::= token | quoted-string
parameter      ::= token '=' argument
unique-name    ::= identifier ( '.' identifier )*
symbolic-name  ::= token('.'token)*
package-name   ::= unique-name
path           ::= path-unquoted | ('"' path-unquoted '"')
path-unquoted  ::= path-sep | path-sep? path-element (path-sep path-
                     element)*
path-element   ::= [^/"\#x0D#x0A#x00]+
path-sep       ::= '/'
url            ::= token schema-sep  path-element  path-unquoted?
schema-sep     ::= token ':' | token '://'
ef-value       ::= boolean | number | float | token | quoted-string |
                     ef-enum | ef-map | ef-interval
boolean        ::= 'true' | 'Y' | 'false' | 'N'
float          ::= <see 64-bit IEEE float format>
ef-enum        ::= '{' token (',' token)* '}'
ef-map         ::= '{' ef-pair (',' ef-pair)* '}'
ef-pair        ::= identifier ':' token
ef-interval    ::= ('(' | '[') int-min (',' int-mid)? ',' int-max
                      ( ']' | ')' )
int-min,
int-mid,
int-max        ::= number | float | token
```

See details of individual headers in section 3.1.3 for constraints on the values applicable.

The following expressions are used throughout the specifications:

Version

```
  version   ::= major( '.' minor ( '.' micro ( '.' qualifier )? )? )?
  major     ::= number
  minor     ::= number
  micro     ::= number
  qualifier ::= ( alphanum | '_' | '-' )+
```

A version token must not contain any whitespace. The default value for a version is 0.0.0.

Version Range

```
  version-range ::= interval | atleast
  interval      ::= ( '[' | '(' ) floor ',' ceiling ( ']' | ')' )
  atleast       ::= version
  floor         ::= version
  ceiling       ::= version
```

Example	Predicate
`[1.2.3, 4.5.6)`	`1.2.3 <= x < 4.5.6`
`[1.2.3, 4.5.6]`	`1.2.3 <= x <= 4.5.6`
`(1.2.3, 4.5.6)`	`1.2.3 < x < 4.5.6`
`(1.2.3, 4.5.6]`	`1.2.3 < x <= 4.5.6`
`1.2.3`	`1.2.3 <= x`

A.5 Manifest headers

This section provides the syntactical definitions of individual CoSi manifest headers.

A.5.1 Simple headers

```
  Bundle-Classpath     ::= entry ( ',' entry )*
        entry ::= path | '.'
  Bundle-Description   ::= quoted-string
  Bundle-ExtraFunc     ::= token ( ',' token )*
  Bundle-Name          ::= symbolic-name
  Bundle-Provider      ::= quoted-string
  Bundle-Version       ::= version
  Control-Class        ::= symbolic-name
  Cosi-Version         ::= number
  ExtraFunc-Registry   ::= path | url
```

A.5.2 Feature headers

Provide-Services

```
  Provide-Services     ::= export ( ',' export )*
```

```
export               ::= interfaces ( ’;’ parameter )*
interfaces           ::= interface ( ';' interface )*
interface            ::= unique-name
```

The header consists of *exports* separated by a comma. Each export defines an service interface and parameters of this interface. Each import can contain more interfaces, but only one set of parameters. In that case, same set of parameters is assigned to each interface.

CoSi recognizes the following parameters: `name, version, extrafunc.`

Require-Services

```
Require-Services     ::= import ( ',' import )*
import               ::= interfaces ( ';' parameter )*
interfaces           ::= interface ( ';' interface )*
interface            ::= unique-name
```

The header consists of *imports* separated by a comma. Each import defines an service interface and parameters of this interface. Each import can contain more interfaces, but only one set of parameters. In that case, same set of parameters is assigned to each interface.

CoSi recognizes the following parameters: `name, versionrange, optional, extrafunc`, and the following deprecated parameters: `bundle-name, bundle-provider, bundle-versionrange.`

Generate-Events

```
Generate-Events      ::= export ( ’,’ export )*
export               ::= messages ; message-type-parameter ( ’;’
                            parameter )*
messages             ::= message ( ';' message )*
message              ::= unique-name
message-type-parameter ::= 'type=' provided-type-unique-name
```

The “message-type-parameter” parameter is mandatory. The header consists of *exports* separated by a comma. Each export contains at least one message name and type. Export can contain more message names, but only one message type. In that case each message has the same type.

Consume-Events

```
Consume-Events ::= import ( ’,’ import )*
import               ::= messages ; message-type-parameter ( ’;’
                     parameter )*
messages             ::= message ( ';' message )*
message              ::= unique-name
message-type-parameter ::= 'type=' provided-type-unique-name
```

The “message-type-parameter” parameter is mandatory. The header consists of *imports* separated by a comma. Each import contains at least one message name and type. Export can contain more message names, but only one message type. In that case each message has the same type.

Provide-Attributes

```
Provide-Attributes   ::= export ( ',' export )*
export               ::= attributes ; attribute-type-parameter ( ';'
                           parameter )*
attributes           ::= attribute ( ';' attribute )*
attribute            ::= unique-name
attribute-type-parameter ::= 'type=' provided-type-unique-name
```

The "attribute-type-parameter" parameter is mandatory. The header consists of *exports* separated by a comma. Each export contains at least one attribute name and type of the attribute. The export can contain more attribute names, but only one attribute type. In that case each attribute has the same type.

Require-Attributes

```
Require-Attributes   ::= import ( ',' import )*
import               ::= attributes ; attribute-type-parameter ( ';'
                           parameter )*
attributes           ::= attribute ( ';' attribute )*
attribute            ::= unique-name
attribute-type-parameter ::= 'type=' provided-type-unique-name
```

The "attribute-type-parameter" parameter is mandatory. The header consists of *inports* separated by a comma. Each import contains at least one attribute name and type of the attribute. The import can contain more attribute names, but only one attribute type. In that case each attribute has the same type.

Provide-Types

```
Provide-Types ::= Provide-Interfaces
```

Has same syntax as Provide-Services definition.

Require-Types

```
Require-Types ::= Require-Interfaces
```

Has same syntax as Require-Services definition.

References

[EJB3] Sun Microsystems, Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements. 2006

[CCM4] Object Management Group, CORBA Component Model Specification, Version 4.0. 2006

[SF25] Rod Johnson et al., Spring Framework, version 2.5. 2007

[OR4] The OSGi Alliance, OSGi Service Platform Core Specification, Release 4.1. 2007

[Szy02] Clemens Szyperski, Component Software. ACM Press, Addison-Wesley 2002

[Bac00] Felix Bachmann and others, Volume II: Technical Concepts of Component-Based Software Engineering. Software Engineering Institute, Carnegie Mellon University 2000

[SOFA] T.Bures, P.Hnetynka, F.Plasil, SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. Proceedings of SERA2006, IEEE CS 2006

[Hei01] George T. Heineman and William T. Councill, Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley 2001

[Waj07] Břetislav Wajtr, Implementace jednoduchého komponentového modelu (in Czech; Implementation of a Simple Component Model). 2007