The Deployment Component

Table of Contents

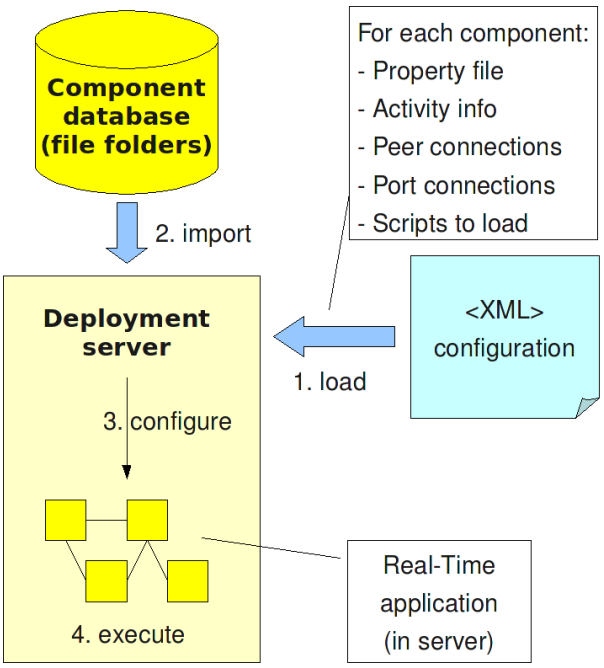
1. Introduction

This document describes the Orocos DeploymentComponent for loading and configuring other components using an Orocos script or XML file. This component can only load components into the same process.

1.1. Principle

Each Orocos component can be compiled as a shared, dynamic loadable library. Each such library can define a special function which will allow the DeploymentComponent to create new instances of a component type. This principle is analogous to the plugin mechanism found in web browsers or other desktop applications.

A common usage scenario of the DeploymentComponent goes as follows. An initial Orocos application is created which contains only the DeploymentComponent and the TaskBrowser. When the application is started, the TaskBrowser prompts for commands which can be given to the DeploymentComponent.



Components are located on your disk using the 'import' statement, loaded using 'loadComponents' and configured using 'configureComponents'. These three steps can be described in an XML file format, a script or using the command prompt.

Figure 1. Component Deployment Overview

Figure 1, “Component Deployment Overview” shows the basic steps. An XML file contains instructions for the DeploymentComponent where to look for components ('import statements'), which component types to create, which name they must be given and how their internal thread is configured (priorities, periods,...). Furthermore this file describes the network interconnections between all components and how data must be relayed from one component to another. The `loadComponents("file.xml")` method reads this file, looks up the components, creates them and stores the configuration parameters. One can apply the configuration (threads, properties, data connections,...) by calling `configureComponents()`. After this step, the components (and the application as a whole) can be started with `startComponents()`. In order to do these steps at once, you can just write `kickStart("file.xml")` or in case of a script, `runScript("file.ops")`

The configuration does not need to be stored in XML format. One can apply the same configuration by using the scripting methods of the DeploymentComponent at the console prompt, or by listing them in an Orocos script.

1.2. The Orocos Deployer Application

The Orocos Component Library provides a number of ready to use applications for loading and starting components using the DeployementComponent.

The main application is the **deployer-<target>** program, where *<target>* is replaced by the Operating System target (OROCOS_TARGET) for which you want to load

components, for example **deployer-gnulinux**. The program can take an optional argument `--start <filename>` which describes the components to load and is used to kick-start the application. The XML and script specifications are described below. When the application is started, the TaskBrowser is presented to the user for receiving interactive commands. The name of the DeploymentComponent is by default 'Deployer'. In order to change this name, use for example **deployer-gnulinux NewDeployerName**. See also **deployer-<target> --help** for an overview of the options.

Note

In case you set the OROCOS_TARGET environment variable to the target you want to use (for example "gnulinux"), you can also start the **deployer** command (which is a shell script), which will in turn start the **deployer-$OROCOS_TARGET** program. If no OROCOS_TARGET has been set, it will refuse to start.

Similar scripts are available for **rttscript**, **cdeployer**, **ctaskbrowser** and **deployer-corba**.

There are four related programs to the deployer application.

- **rttscript-<target>**: like above but does not show a TaskBrowser prompt. When it is finished deploying, it undeploys and quits. You'll need to use a crafted .ops script to hold-off the quiting until your applications needs to. Useful for doing little tasks from the command line, unit tests or another script.

- **cdeployer-<target>**: like above but starts the CORBA enabled non-interactive deployer application. You are not presented with a TaskBrowser prompt, but the cdeployer tries to connect to a CORBA Naming Service, and if that fails, prints the IOR to a file and to the screen.

- **ctaskbrowser-<target> ComponentName|IOR**: Connects to a remote component (like the cdeployer above) using the CORBA *IOR* address or using the CORBA Naming Service using *ComponentName*.

- **deployer-corba-<target>**: Combines the cdeployer and deployer applications. It presents the TaskBrowser console and sets up a CORBA server. It can thus be commanded locally and accessed over a network. If you quit the TaskBrowser prompt, the application exits.

The complete list of options for the deployer, cdeployer and deployer-corba programs are:

- **--help** Show program usage

- **--start xml-or-script-file** (also -s) Deploy from an *xml-file (.xml or .cpf)* or *script-file (.ops or .osd)*. This option may be given multiple times, in which case the xml and script files will be processed in the same order as on the command line.

- **--log-level level** (also -l) Sets the Orocos log level to *level*. The level parameter should be one of: Never, Fatal, Critical, Error, Warning, Info, Debug, or Realtime. The parameter is case-insensitive. Warning: this *overrides* the ORO_LOGLEVEL environment variable.

- **--no-consolelog** Turn off logging to the console (will still log to 'orocos.log')

- **--daemon** (also -d) run the deployer as a background process. Will not open up a TaskBrowser prompt.

- **--DeployerName deployer-name** Name of deployer component (the --Deployer-Name flag is optional)

Additionally, any CORBA options can be passed through these programs by adding a "--" command line option, followed by the CORBA-specific options.

Some examples are

```
deployer-corba --log-level warning -s myfile.xml
```

Sets the Orocos log level to `warning` and deploys file `myfile.xml`

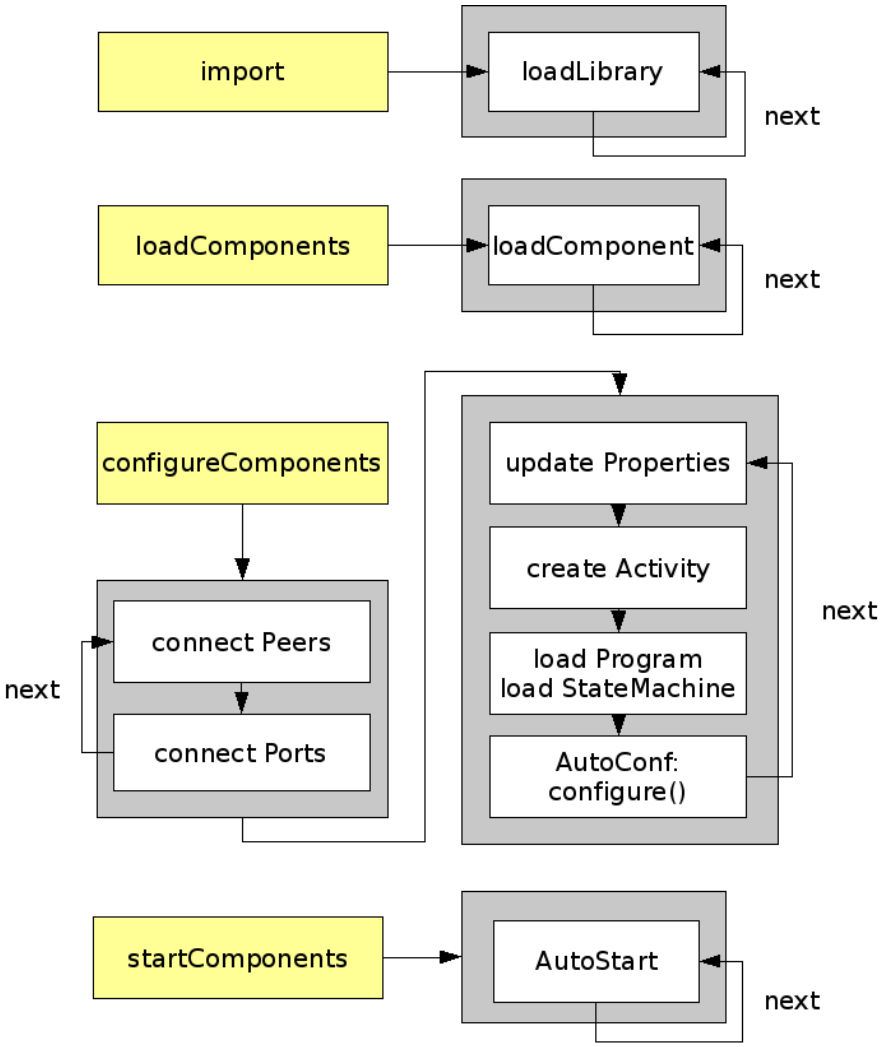
```
deployer-corba -l fatal --no-consolelog -s leftfile.xml LeftDeployer
```

Sets the Orocos log level to `fatal`, turns off all logging to console, names the deployer `LeftDeployer` and deploys file `leftfile.xml`

```
deployer-corba -l fatal --no-consolelog -s leftfile.xml LeftDeployer
-- -ORBInitRef NameService=corbaloc:iiop:me.mine.home:2809/NameService -
ORBFooBar 1
```

As with the previous example, and also passes some options through to the CORBA layer.

2. Configuration Procedure



The Deployment component API consists of import, loadComponents, configure-Components and startComponents.

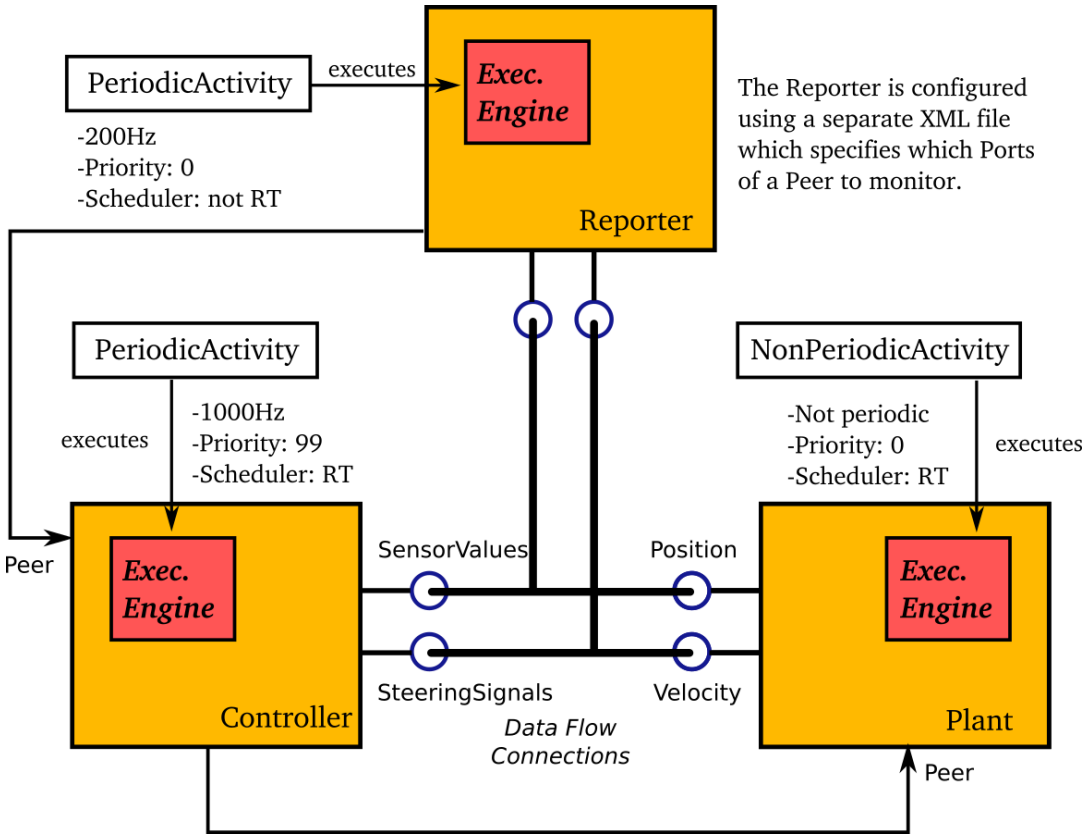
Figure 2. Deployment Procedure

The configuration format defines the instructions one can use to load and configure Orocos components. One can divide the instructions in three groups:

- Where to look for component libraries and plugins
- Which components to create and with which name
- How each component is setup

Let's demonstrate this principle with a simple application example as shown in Figure 3, “Deployment Example Application”. We want to setup an application with three components: a Reporting component, a 'Controller' and a 'Plant'. The Plant component provides access to the hardware, the Controller component contains the con-

trol algorithm. The Reporting component is here to sense the values exchanged and write them to a file.



A Reporter component monitors communication between Plant and Controller. The Deployment component itself is not shown.

Figure 3. Deployment Example Application

2.1. Where to look for component libraries

The `path` and `import` statements are the two ways to specify where components can be found, and which component libraries to import.

Imagine that you have this directory structure:

```
/opt/robot                    : your orocos install path (lib, include,
etc)
/opt/robot/lib/orocos/        : orocos installed components
                      /plugins : services and other plugins
                      /types   : typekits and transports
```

And that you have a project 'robot-13' which is installed there as well, but in a sub-directory of the `/opt/robot/lib/orocos` directory:

```
/opt/robot/lib/orocos/robot-13         : robot-13 components
                      /robot-13/plugins : robot-13 services and other
plugins
                      /robot-13/types   : robot-13 typekits and transports
```

The RTT Plugins manual describes this typical directory structure more in detail.

The `path` function extends the default search path with new directories to look for components. In addition, it imports every component library found in that directory, but *without recursing into sub directories*. It does not cause any component to be created, but allows the DeploymentComponent to know where the component libraries are located. This function may be called for multiple paths, or provide them in a colon or semi-colon separated list.

In XML, the path statement looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <!-- ....  -->

  <!-- Note: capital 'P': -->
  <simple name="Path" type="string"><value>/opt/robot/lib/orocos</value></
simple>

</properties>
```

The script method equivalent is:

```
  // note: small 'p':
  path("/opt/robot/lib/orocos")
```

Each component library (.so, .dll,...) in the directory `/opt/robot/lib/orocos` is imported. If this directory contains a `plugins` or `types` subdirectory, the libraries in these directories are imported as well. Once you installed multiple component libraries in subdirectories of your path, you must use the `Import` statement to load these. In addition, when you use the ROS packaging system, you can use `Import` in order to load the components from a ros package's `lib/orocos` directory, and all its dependencies. In that case, only the ROS_PACKAGE_PATH environment variable needs to be set.

In XML, the import statement looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <!-- ....  -->

  <!-- Note: capital 'I': -->
  <simple name="Import" type="string"><value>robot-13</value></simple>

</properties>
```

The script method equivalent is:

```
  // note: small 'i':
  import("robot-13")
```

All component libraries found in `<path>/robot-13` (or the ROS package 'robot-13') and their plugins/types are loaded because of this statement. The import and path statements *only allow you to load Orocos plugin or component libraries*. In case the import statement contains a path to an existing library file, that file will be loaded directly instead of looking it up in the search paths.

See the Plugin manual for creating plugin libraries or the end of this manual for creating component libraries. Regular libraries (like libfoo.so or win32.dll,...) can not be loaded. If a library contains one or more Orocos components, the contained component types become available in the next step.

To see the effects of the import function, the available types can be queried by invoking the `displayComponentTypes` (script) method:

```
 (type 'ls' for context info) :displayComponentTypes()
      Got :displayComponentTypes()
 = I can create the following component types:
   TaskContext
   OCL::ConsoleReporting
   OCL::FileReporting
   OCL::HelloWorld
   Robot13::Controller
   Robot13::Diagnostics
(void)
```

Summarized:

- `path` pre-loads component libraries and sets the search path for subdirectories
- `import` loads component libraries from subdirectories in the search path OR a specific file directly
- `displayComponentTypes` shows which components have been found.

2.2. Including other XML files

In order to manage your XML files, one XML file can include another with the 'Include' directive. The include directive may occur at any place in the XML file (but under <properties>) and will be processed as-if the included file is inserted at that point.

Warning

This option is new and experimental and may change in meaning and/or name in the future. When using the Xerces XML parser in Orocos, you may also want to use the standard XML way for including external documents, as documented on the Orocos Wiki.

In XML, the include statement looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <!-- ....  -->

  <simple name="Include" type="string"><value>default-imports.xml</
value></simple>
  <simple name="Include" type="string"><value>default-components.xml</
value></simple>

</properties>
```

2.3. Which components to create and with which name

Import makes components available, but does not create an *specific instance* yet. In order to add a component of a given type to the current application, use the `loadComponent` function:

In XML, the loadComponent statement of a reporting component would look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <!-- ... import statements locate Orocos reporting library ...  -->
  <simple name="Import" type="string"><value>/usr/local/lib/orocos</
value></simple>

  <struct name="Reporter" type="OCL::FileReporting">
  </struct>

</properties>
```

This line causes the DeploymentComponent to look up the OCL::FileReporting type, and if found, creates a component of that type with the name "Reporter". This component is added as a peer component to the DeploymentComponent such that it becomes immediately available to the application. This step can be repeated any number of times with any number of components or names.

Alternatively, the type may be a filename if that file contains only one component, which is exported using the `ORO_CREATE_COMPONENT` macro (see below).

The script method equivalent is:

```
  loadComponent("Reporter", "OCL::FileReporting")
```

2.4. How each component is setup

Now that one or more component instances are created, you can configure them by connecting components, assigning threads, configuration values and program scripts. Again, you can do this using XML or the scripting language.

Below is an example of about all options you can use. They are explained in the sections below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <simple name="Import" type="string"><value>/usr/local/lib/orocos</
value></simple>

  <!-- You can set per data flow connection policies -->
  <struct name="SensorValuesConnection" type="ConnPolicy">
    <!-- Type is 'shared data' or buffered: DATA: 0 , BUFFER: 1 -->
    <simple name="type" type="short"><value>1</value></simple>
    <!-- buffer size is 12 -->
    <simple name="size" type="short"><value>12</value></simple>
  </struct>
  <!-- You can repeat this struct for each connection below ... -->
```

```
  <struct name="Reporter" type="OCL::FileReporting">

    <struct name="Activity" type="Activity">
      <simple name="Period" type="double"><value>0.005</value></simple>
      <simple name="Priority" type="short"><value>0</value></simple>
      <simple name="Scheduler" type="string"><value>ORO_SCHED_OTHER</
value></simple>
    </struct>

    <simple name="AutoConf" type="boolean"><value>1</value></simple>
    <simple name="AutoSave" type="boolean"><value>1</value></simple>

    <simple name="LoadProperties" type="string"><value>file-
reporting.cpf</value></simple>

    <struct name="Peers" type="PropertyBag">
      <simple type="string"><value>Controller</value></simple>
    </struct>
  </struct>

  <struct name="Controller" type="ControllerType">

    <struct name="Activity" type="Activity">
      <simple name="Period" type="double"><value>0.001</value></simple>
      <simple name="Priority" type="short"><value>99</value></simple>
      <simple name="Scheduler" type="string"><value>ORO_SCHED_RT</value></
simple>
    </struct>

    <!-- loads the 'scripting' service (aka plugin) in this component -->
    <simple name="Service" type="string"><value>scripting</value></simple>

    <simple name="AutoConf" type="boolean"><value>1</value></simple>
    <simple name="AutoStart" type="boolean"><value>1</value></simple>
    <simple name="AutoConnect" type="boolean"><value>1</value></simple>

    <!-- This section allows to define properties without using a file
 (see below)
         These properties can be overriden in the property files below. --
>
    <struct name="Properties" type="PropertyBag">
      <simple name="K" type="double"><value>1.0</value></simple>
    </struct>
    <!-- Note: difference between 'PropertyFile' and
 'UpdateProperties' (see below) -->
    <simple name="PropertyFile" type="string"><value>controller-main.cpf</
value></simple>
    <simple name="UpdateProperties" type="string"><value>controller-
opts.cpf</value></simple>

    <struct name="Ports" type="PropertyBag">
      <!-- Note: the value is the name of the connection of which this
 port gets part.
           All ports that share the same connection name are connected to
 each other
           The connection policy for SensorValuesConnection was defined
 above. If no
    policy is given, the default (DATA, LOCK_FREE) is used.
      -->
      <simple name="SensorValues"
 type="string"><value>SensorValuesConnection</value></simple>
      <simple name="SteeringSignals"
 type="string"><value>DriveConnection</value></simple>
    </struct>
```

```
    <struct name="Peers" type="PropertyBag">
      <simple type="string"><value>Plant</value></simple>
    </struct>

    <simple name="RunScript" type="string"><value>controller-program.ops</
value></simple>
    <simple name="RunScript" type="string"><value>controller-states.ops</
value></simple>
  </struct>

  <struct name="Plant" type="PlantType">
    <struct name="Activity" type="Activity">
      <simple name="Priority" type="short"><value>0</value></simple>
      <simple name="Scheduler" type="string"><value>ORO_SCHED_RT</value></
simple>
    </struct>
    <simple name="AutoStart" type="boolean"><value>1</value></simple>
    <struct name="Ports" type="PropertyBag">
      <simple name="Position"
 type="string"><value>SensorValuesConnection</value></simple>
      <simple name="Velocity" type="string"><value>DriveConnection</
value></simple>
    </struct>
  </struct>
</properties>
```

2.4.1. Thread, period, priority and scheduler.

The first section of all three components sets up the active behaviour of the component in the `Activity` element.

```
    <struct name="Activity" type="Activity">
      <simple name="Period" type="double"><value>0.005</value></simple>
      <simple name="Priority" type="short"><value>0</value></simple>
      <simple name="Scheduler" type="string"><value>ORO_SCHED_OTHER</
value></simple>
    </struct>
```

Both have periodic activities, which run with a given period, priority and in a scheduler. The Controller and Plant run in a real-time scheduler, the Reporter doesn't. The activities are created and attached to each component during the `configureComponents()` step of the DeploymentComponent. Possible types of activities are

- `PeriodicActivity`,
- `Activity` (the standard one),
- `SequentialActivity` and
- `SlaveActivity`.

The latter allows a component to be executed by a master component. You can specify a master component using the `Master` simple element in the `Activity` struct. The DeploymentComponent makes slaves automatically a peer of their master, but does nothing more. Ie, the code in the master's updateHook() must call trigger on each of its slaves that are peers.

2.4.2. Loading Services or Plugins.

You can load any number of plugins into a component. A plugin may also add a Service object to a component's interface, but this is optional.

```
<!-- loads the 'scripting' service in this component -->
<simple name="Service" type="string"><value>scripting</value></simple>

<!-- loads the 'trajectory' plugin in this component -->
<simple name="Plugin" type="string"><value>trajectory</value></simple>
```

The `Service` or `Plugin` element may occur any number of times in the component struct to list a specific service or plugin that must be loaded in that component. For example, in order to execute a script in your component, you may load the 'scripting' service. Or in order to serialize its properties to XML, you'll need the 'marshalling' service. These services add new functions to your component which provide that functionality.

A service promisses that it is available as a Service object in the component's interface. A plugin doesn't have this obligation, and can have any desired effect on your component.

You can check the available services or plugins (ie discovered by the DeploymentComponent) with '.services' or '.plugins' and load a service from the TaskBrowser prompt *in the current visited component* with

```
.provide
     <servicename>
```

. The Deployer has the equivalent function which looks like this:

```
loadService("Reporter","scripting")
```

Where Reporter must be a peer of the Deployer.

2.4.3. Auto-Configuration and Auto-Starting components.

The next section of the Controller contains the `AutoConf` and `AutoStart` elements.

```
<simple name="AutoConf" type="boolean"><value>1</value></simple>
<simple name="AutoStart" type="boolean"><value>1</value></simple>
<simple name="AutoConnect" type="boolean"><value>1</value></simple>
```

If `AutoConf` is set to 1, the DeploymentComponent will call the component's configure() method during `configureComponents()`, after the properties are loaded. If `AutoStart` is set to 1, the component's start() method will be called during `startComponents()`. By default `AutoConf` and `AutoStart` are 0 (off).

There is no literal alternative for AutoConf in scripting. Just use the `configure()` operation of your component in order to configure it:

```
  Controller.configure()
  Controller.start()
```

2.4.4. Connecting Data Ports

The `Ports` struct describes which ports of this component participate in which data flow connection.

```
    <struct name="Ports" type="PropertyBag">
      <simple name="Position"
 type="string"><value>SensorValuesConnection</value></simple>
      <simple name="Velocity" type="string"><value>DriveConnection</
value></simple>
    </struct>
```

So for each element in this struct, the name of the element is the port name, and the value is the name of the connection it belongs to. Ports with equal *connection names* are connected to each other. Ports which are not listed will not be connected to anything. If ports of different data types are being connected, the configuration phase will fail. You can tune each connection using a struct of type `ConnPolicy` with the name of the connection. The allowed fields in this struct are the same as in the C++ API, see ConnPolicy.

```
  <!-- You can set per data flow connection policies -->
  <struct name="SensorValuesConnection" type="ConnPolicy">
    <!-- Type is 'shared data' or buffered: DATA: 0 , BUFFER: 1 -->
    <simple name="type" type="short"><value>1</value></simple>
    <!-- buffer size is 12 -->
    <simple name="size" type="short"><value>12</value></simple>
  </struct>
  <!-- You can repeat this struct for each connection below ... -->
```

In this example, the SensorValuesConnection is configured, which is used to connect the Controller's SensorValues port with the Plant's Position port.

Looking at the Ports section of the Controller above, it has two data ports listed (SensorValues and SteeringSignals), which are added to two connection objects. These connections show up in the Plant component's `Ports` section as well. And it shows that the SensorValues Port is connected to the Position Port and the SteeringSignals Port is connected to the Velocity Port. If other component's ports in the same file refer to the same connection object, the ports are connected to each other by the DeploymentComponent during the `configureComponents()` step.

The `AutoConnect` element indicates if the component's data ports should be automatically connected to peer ports which have the same name and type. This flag is read during the `configureComponents()` step of the DeploymentComponent. Both components must have the `AutoConnect` element set to 1 *and one must be peer of the other* in order to trigger automatic connection of ports. It is possible that a port is connected to one component using the Ports struct and to another component using the `AutoConnect` flag. If an automatic port connection fails, the configura-

tion procedure will not fail and just continue. An error message may be logged. By default, `AutoConnect` is 0 (off).

Note

AutoConnect is only useful for simple applications, use the explicit 'Ports' connection method to connect different named ports to each other !

In scripting, you can use the `ConnPolicy` struct for connecting ports. For example:

```
  var ConnPolicy cp
  cp.type = BUFFER
  cp.size = 10 // buffer of size 10
  connect("Plant.Position", "Controller.SensorValues", cp )
```

You may re-use the 'cp' object multiple times for different connections. Streams can be created likewise, with the `stream` operation of the deployer, which only takes a port and a connection policy as argument.

2.4.5. Setting Properties (component parameters)

The `Properties` struct allows to configure a component's properties from the main XML file. These values can be overridden by the listed property files:

```
  <!-- You can repeat this struct for each connection below ... -->
    <struct name="Properties" type="PropertyBag">
      <simple name="K" type="double"><value>1.0</value></simple>
    </struct>
```

The `PropertyFile` element specifies from which XML file each component is configured and this file *must* contain values for all properties of the component.

In case you only want to update part of the properties, use the `UpdateProperties` element.

```
    <simple name="PropertyFile" type="string"><value>controller-main.cpf</
value></simple>
    <simple name="UpdateProperties" type="string"><value>controller-
opts.cpf</value></simple>
```

Finally, it is also possible to load and create new properties from a file using `Load-Properties` the Reporting component requires this for example:

```
    <simple name="LoadProperties" type="string"><value>file-
reporting.cpf</value></simple>
```

You can use any number or combination of these elements. The order is respected. The properties are read during the `configureComponents()` step of the DeploymentComponent. When the `AutoSave` property is turned on, the listed property file will be saved again with the values of the Component, just before the Component is `cleanup()`.

In scripting, you can use the `marshalling` service in order to do the property loading for you. For example:

```
loadService("MyComponent","marshalling")
MyComponent.marshalling.readProperties("file.cpf")
```

Every component that needs to read/write properties from a file needs the marshalling service. You can't use the marshalling service of the Deployment Component, since that service would read/write the properties of the Deployment Component itself.

2.4.6. Setting up peer-to-peer relations

The last section of the Reporter component lists its peers.

```
  <struct name="Peers" type="PropertyBag">
    <simple type="string"><value>Controller</value></simple>
  </struct>
```

The Reporter has one peer, the Controller, which allows the Reporter component to scan and use the interface of the Controller component. For example, it will discover which ports Controller exposes and be able to create connections to them, without the need of a supervisor to do so.

The Controller component has the Plant as peer, which means it can query and control it. For example, use its services, start and stop it etc.

2.4.7. Loading and Running Orocos Program Scripts

Note

This section is for starting scripts from the XML file. In case you want to use a script directly (or after an XML file), you can use the -s option of the deployer to let it execute that script.

The Controller has at the end two additional `RunScript` elements describing which script files must be loaded and executed into that component.

```
    <simple name="RunScript" type="string"><value>controller-program.ops</
value></simple>
    <simple name="RunScript" type="string"><value>controller-states.ops</
value></simple>
```

Any number of scripts can be loaded and they are loaded in the order of the XML file. Each script may contain any number of statements, functions, program scripts or state machines. Running these scripts is again done during the `configureComponents()` step.

If you want to have a program or statemachine started you need to do so at the end of the script file itself, by adding

```
programname.start()
statemachine_instance.activate()
statemachine_instance.start()
```

statements. Be aware that this is done during the configuration phase of your components, so before updateHook() is executed. You are however allowed to start your component from the script by merely calling

```
  start()
```

at the right place of your script.

You may choose to implement the whole deployment scenario in such a script, instead of the XML file presented in this manual. In that case, you need to load this script in the Deployer itself using the `-s filename.ops` command line option, or using a small XML file that only contains this code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <struct name="Deployer" type="PropertyBag">

    <!-- set a period -->
    <struct name="Activity" type="Activity">
      <simple name="Period" type="double"><value>0.01</value></simple>
    </struct>

    <!-- run a script -->
    <simple name="RunScript" type="string"><value>scriptfile.ops</value></
simple>
  </struct>
</properties>
```

It is advised to set a period for the activity of a component executing scripts, since scripts need periodic execution in case they have to wait for an operation to complete. Alternatively, you can set the period at the top of your script file by adding the statement:

```
  setPeriod(0.01)
```

instead of specifying it in the XML file.

2.5. CORBA extensions

The deployer XML format allows two CORBA specific boolean properties, which are optional: `Server` (defaults to '0') and `UseNamingService` (defaults to '1'). These properties are only used when you use the CORBA enabled **cdeployer-<target>** or **deployer-corba-<target>** applications.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <!-- ... -->

  <struct name="Reporter" type="OCL::FileReporting">

    <!-- CORBA specific extensions -->
    <simple name="Server" type="boolean"><value>1</value></simple>
    <simple name="UseNamingService" type="boolean"><value>1</value></
simple>
```

```
  </struct>

</properties>
```

By default, only the 'Deployer' starts as a CORBA server. You can have other components to start as a server as well by setting the `Server` property to 1. By default, the component will try to use the CORBA Naming Service to register its name. If this is not wanted, set the `UseNamingService` property to 0.

The script method equivalent of the above XML construct is:

```
  server("Reporter", true)
```

Which will create a CORBA server for the Reporter peer, after the Reporter was loaded with loadComponent().

2.6. Connecting to CORBA components

The corba enabled deployers allow to create a proxy for a remote component using the name service, the IOR or the IOR file.

Say you have a remote Orocos component with the name 'MyComponent'. It was created in one corba enabled deployer application with the `Server` property set to 1. You can connect to it from another deployer application by using the XML syntax:

```
<!-- Uses CORBA Naming Service to lookup 'Mycomponent' -->
<struct name="MyComponent" type="CORBA">
</struct>

<!-- Uses IOR file to lookup 'Mycomponent' -->
<struct name="MyComponent.ior" type="IORFile">
</struct>

<!-- Uses literal IOR to lookup 'Mycomponent' -->
<struct name="IOR:...." type="IOR">
</struct>
```

Which will make this component available in your current application, using the same name as the original. This also works for the scripting deployer command 'loadComponent'. For example, you can type in the TaskBrowser:

```
  loadComponent("MyComponent", "CORBA")
  loadComponent("MyComponent.ior", "IORFile")
  loadComponent("IOR:.....", "IOR")
```

which allows to quickly connect to a remote component once you can copy/paste the IOR into the console.

3. Setting up a deployable component library

This section explains how to prepare a component library for deployment. It is demonstrated with an example.

Note

The `orocreate-pkg` script of OCL does all the setup work for you. This section is given for reference use only.

3.1. Additional Code

There exist three C macros for preparing a component library. The simplest way is when the resulting library will contain only one component type. Assume we have written the HelloWorld component (in the OCL C++ namespace) which is compiled in the `orocos-helloworld.so` library. The following code is added to HelloWorld.cpp:

```
#include "HelloWorld.hpp"
#include <ocl/Component.hpp>

/* ... Hello World implementation file ... */

// You must specify the namespace:
ORO_CREATE_COMPONENT( OCL::HelloWorld )
```

This macro inserts a function into the library which will allow the DeploymentComponent to create OCL::HelloWorld components.

In case multiple components are defined in the same library, two other macros must be used. One macro for each component type and one macro once for the whole library. Say your library has components NS::`ComponentX` and NS::`ComponentZ` in namespace NS. In order to export both components, you could write in `ComponentX.cpp`:

```
#include "ComponentX.hpp"
#include <ocl/Component.hpp>

/* ... ComponentX implementation file ... */
// once:
ORO_CREATE_COMPONENT_LIBRARY()
// For the ComponentX type:
ORO_LIST_COMPONENT_TYPE( NS::ComponentX )
```

and in `ComponentY.cpp` the same but without the ORO_CREATE_COMPONENT_LIBRARY macro:

```
#include "ComponentY.hpp"
#include <ocl/Component.hpp>

/* ... ComponentY implementation file ... */

// For the ComponentY type:
ORO_LIST_COMPONENT_TYPE( NS::ComponentY )
```

For each additional component in the same library, the ORO_LIST_COMPONENT_TYPE macro is added. It is allowed to put all the ORO_LIST_COMPONENT_TYPE macros in a single file.

Note

You may not link multiple libraries that use ORO_CREATE_COMPONENT, since only one of the component types will be found.

Note

ORO_CREATE_COMPONENT_LIBRARY() replaces the pre-2.3.0 ORO_CREATE_COMPONENT_TYPE() macro. The old macro is still kept for backwards compatibility, both versions have the exact same result.

3.2. Compiling and linking a component library

In order to have a working library, care must be taken of the compilation flags. You may compile your library static or shared. But a static library will not be dynamically loadable. In the final executable the DeploymentComponent will be able to find the linked in components and setup the application using the XML file.

Important

The macros need some help to figure out if you are compiling a shared or static library. You need to define the *RTT_COMPONENT* macro (see below) when compiling for a shared library. If this macro is not defined, it is assumed that you are compiling for a static library.

The compilation flag of a component for a shared library might look like:

```
CFLAGS= -O2 -Wall -fPIC -DRTT_COMPONENT
LDFLAGS= -fPIC
```

The compilation flag of a component for a static library lacks both options :

```
CFLAGS= -O2 -Wall
LDFLAGS=
```

Note

If you use CMake with the UseOrocos.cmake macros, you don't need any of this manual setup. The Orocos macros set the right flags for you.