
The Reporting Component

Copyright © 2006,2007,2008,2009 Peter Soetens, FMTC

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation, with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of this license can be found at <http://www.fsf.org/copyleft/fdl.html>.

Table of Contents

1. Introduction	1
1.1. Principle	1
2. Setup Procedure	2
2.1. Property Configuration File	3
2.2. ReportData section	4
2.3. Reading the configuration file	4
3. Scripting commands	5
4. Forcing data reporting (snapshot).	5
5. Future work	6

1. Introduction

This document describes the Orocos ReportingComponent for monitoring and capturing data exchanged between Orocos components.

1.1. Principle

Each Orocos component can have a number of data ports. One can configure the reporting components such that one or more ports are captured of one or more peer components. The reporting components can work sample rate based or event based. A number of file format can be selected.

The Reporter can uses buffers in order to log all data it receives or just report the last values in case it is flooded with data. This must be configured for all connections equally.

A common usage scenario of the ReportingComponent goes as follows. An Orocos application is created which contains a reporting component and various other components. The reporting component is peer-connected to all components which must be monitored. An XML file or script command defines which data ports to log of each peer. When the reporting component is started, it reads the ports and writes the exchanged data to a file at a given sample rate or when new data is written.

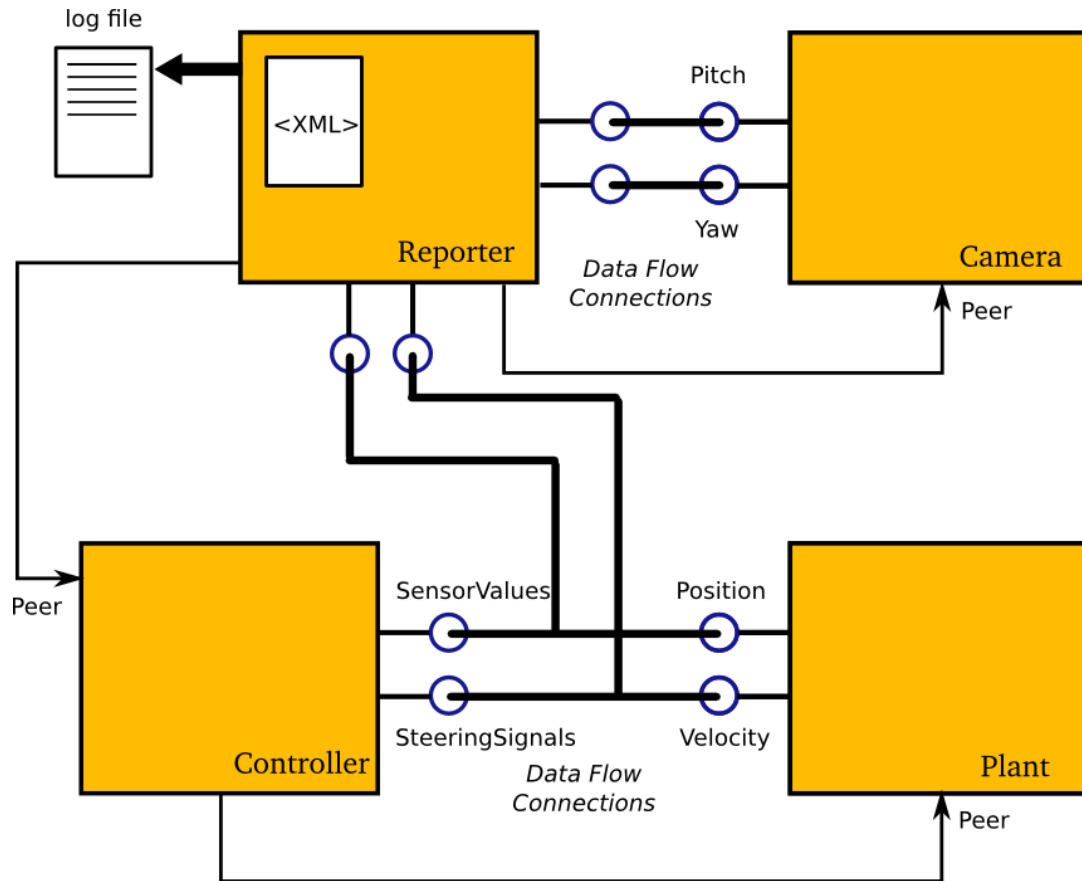


Figure 1. Component Reporting Example

One can not use the ReportingComponent directly but must use a derived component which implements the method of writing out the data. There exists a number variants: FileReporting for writing data to a file and ConsoleReporting which prints the data directly to the screen. The NetcdfReporting writes the NetCDF file format. In order to support other file formats, you can write your own marshaller.

2. Setup Procedure

The ReportingComponent is configured using a single XML file which sets the component's properties and describes which components and ports to monitor.

In order to report data of other components, they must be added as a Peer to the reporting component. The following C++ setup code does this for the example above (Figure 1, "Component Reporting Example"):

```
#include <ocl/FileReporting.hpp>

// ...
OCL::FileReporting reporter("Reporter");
Controller controller("Controller");
Plant plant("Plant");
Camera cam0("Camera");

reporter.addPeer( &controller );
reporter.addPeer( &camera );
```

```
controller.addPeer( &plant );
```

Alternatively, if you use the Deployer, you might add this listing to your application XML file, instead of using the hard-coded C++ setup above:

```
<simple name="Import" type="string"><value>/usr/lib/liborocos-
reporting</value></simple>

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

  <!-- Note: Activity may also be non-periodic -->
  <struct name="Activity" type="Activity">
    <simple name="Period" type="double"><value>0.01</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="AutoStart" type="boolean"><value>0</value></simple>
  <simple name="AutoSave" type="boolean"><value>1</value></simple>
  <simple name="LoadProperties" type="string"><value>reporting.cpf</
value></simple>
  <!-- List all peers (uni-directional) -->
  <struct name="Peers" type="PropertyBag">
    <simple type="string"><value>Controller</value></simple>
    <simple type="string"><value>Camera</value></simple>
  </struct>
```

Note that the AutoSave flag is turned on (this is optional) to save the settings when the Reporter component is cleaned up by the Deployer.

If the Reporter has a periodic activity, it will sample all its input ports and write out the new values. For old values, a '-' (dash) is written by default instead of a value. You can configure this behaviour by using the NullSample property (see below).

If the Reporter's activity is non-periodic (Period omitted or zero), it will only write out a new value when new data arrives on one of the connected ports. Ports that did not get a new value will also output the NullSample.

The values of reported attributes or properties are always printed, and do not cause the triggering of a new data report.

2.1. Property Configuration File

This is an example property file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <simple name="WriteHeader" type="boolean">
    <description>Set to true to start each report with a header.</
description><value>1</value>
  </simple>
  <simple name="NullSample" type="string">
    <description>The characters written to the log to indicate
    that no new data was available for that port during a snapshot().
    As a special value, the string 'last' is interpreted as repeating
    the last value.</description>
    <value>--</value>
  </simple>
  <simple name="Decompose" type="boolean">
```

```
<description>Set to false in order to create multidimensional array in
netcdf</description><value>1</value>
</simple>
<simple name="Synchronize" type="boolean">
  <description>Set to true if the timestamp should be synchronized with
the logging</description><value>0</value>
</simple>
<simple name="WriteHeader" type="boolean">
  <description>Set to true to start each report with a header.</
description><value>1</value>
</simple>
<simple name="ReportFile" type="string">
  <description>Location on disc to store the reports.</
description><value>reports.dat</value>
</simple>

<struct name="ReportData" type="PropertyBag">
  <description>A PropertyBag which defines which ports or components to
report.</description>
  <simple name="Component" type="string">
    <description>Report all output ports of this component.</
description><value>MyPeer2</value>
  </simple>
  <simple name="Port" type="string">
    <description>Report this output port</
description><value>MyPeer.D2Port</value>
  </simple>
  <simple name="Data" type="string">
    <description>Report this property/attribute</
description><value>MyPeer.Hello</value>
  </simple>
</struct>
</properties>
```

If `WriteHeader` is set to true, a header will be written describing the file format layout.

If `Decompose` is set to true, complex data on ports will be decomposed into separate table columns. Otherwise, each port appears as one column. The `NetcdfReporter` requires this to be false, most other reporters are better off if this is true (the default).

The `NullSample` is a string which you can set to last in order to repeat the last value if no new data arrived. This can not be set on a per port basis.

2.2. ReportData section

The `ReportData` struct describes the ports to monitor. As the example shows (see also Figure 1, “Component Reporting Example”), a complete component can be monitored (Camera) or specific ports of a peer component can be monitored. The reporting component can monitor any data type as long as it is known in the Orocos type system. Extending the type system is explained in the Plugin Manual of the Real-Time Toolkit.

2.3. Reading the configuration file

The property file of the reporting component *must* be read with the `loadProperties` script method:

```
marshalling.loadProperties("reporting.cpf")
```

You can not use `readProperties()` because only `loadProperties` loads your `ReportData` struct into the `ReportingComponent`.

With

```
marshalling.writeProperties("reporting.cpf")
```

, the current configuration can be written to disk again.

3. Scripting commands

The scripting commands of the reporting components can be listed using the **this** command on the `TaskBrowser`. Below is a snippet of the output:

```
RTT::Method      : bool reportComponent( string const& Component )
  Add a peer Component and report all its data ports
  Component : Name of the Component
RTT::Method      : bool reportData( string const& Component, string
const& Data )
  Add a Component's Property or attribute for reporting.
  Component : Name of the Component
  Data : Name of the Data to report. A property's or attribute's name.
RTT::Method      : bool reportPort( string const& Component, string
const& Port )
  Add a Component's OutputPort for reporting.
  Component : Name of the Component
  Port : Name of the Port.
RTT::Method      : bool screenComponent( string const& Component )
  Display the variables and ports of a Component.
  Component : Name of the Component
RTT::Method      : void snapshot( )
  Take a new shapshot of all data and cause them to be written out.
RTT::Method      : bool unreportComponent( string const& Component )
  Remove all Component's data ports from reporting.
  Component : Name of the Component
RTT::Method      : bool unreportData( string const& Component, string
const& Data )
  Remove a Data object from reporting.
  Component : Name of the Component
  Data : Name of the property or attribute.
RTT::Method      : bool unreportPort( string const& Component, string
const& Port )
  Remove a Port from reporting.
  Component : Name of the Component
  Port : Name of the Port.
```

4. Forcing data reporting (snapshot).

One can force that all current data ports are sampled and written out. This is independent of the periodic or non periodic execution of the Reporter. There is a method available that is called `snapshot()`. It may be called from real-time contexts. Since a snapshot writes out all data samples, it is not possible to see which ones changed in comparison to the previous snapshot.

In case you *only* want to log data in case of a `snapshot()`, you need to set the `SnapshotOnly` property to 1. Even more when this flag is set, the reporting

component will reduce all connections to being unbuffered during a `configure()` run, such that a `snapshot()` always returns the most recent values.



Note

`SnapshotOnly` is the inverse of the OCL 1.x `AutoTrigger` option. If you had `AutoTrigger` disabled, then enable `SnapshotOnly` in OCL 2.x.

5. Future work

In time, it should become possible to set buffering policies for each connection, or let the deployment component form the connections from components to reporting and let the reporting component pick them up. This would allow for an easier use and configurability.