RT MIDDLEWARE

OpenRTM-aist-0.2.0 Developer's Guide

National Institute of Advanced Industrial Science and Technology (AIST)
Intelligent Systems Research Institute - Task Intelligence Research Group
OpenRTM Project Team

MIDDLEWARE

Contents

Structure of this Book

This book is the official developer's manual of OpenRTM-aist. OpenRTM-aist was developed by the National Institute of Advanced Industrial Science and Technology under the 21st century Robot Challenge Programme called "Development of the key enabling technologies for Robotics" sponsored by the New Energy and Industrial Technology Development Organization (NEDO). While the work toward the standardization of the interface definitions included in the RT-Middleware specification is still in progress, OpenRTM-aist is an open and free-to-use implementation of the RT-Middleware. By using OpenRTM-aist, it becomes easy to build networked robotic systems supported by the distributed objects technology.

OpenRTM-aist is a middlleware which has been especially designed to make the development of robotic systems easier. However, as it is still at an early stage of development, not all the necessary functionalities have been yet implemented. Moreover, we expect that it still contains a significant amount of bugs. From now on, thanks to the feedback provided by all the users, we would like to keep extending it toward an even easier to use middleware.

Chatpter 1 : What is the RT-Middleware ?

In this chapter, we will introduce the basic concepts of the RT-Middleware. We will first explain the reasons that lead us to the development of the RT-Middleware. Also, we will explain what the RT-Middleware is, its concepts, its features as well as its basic architecture.

Chapter 2 : Introduction to RT-Components

In this chapter, we will explain how to install and use OpenRTM-aist, one implementation of the RT-Middleware. For a complete description of how to install all the necessary packages, refer to the section "How to build the dependencies packages" in the appendix.

Chapter 3 : Programming an RT-Component

In this chapter, we will introduce the morphology of the components as defined in OpenRTM-aist and how to execute them. We will also explain how to build components using OpenRTM-aist.

Chapter 4 : Tools in OpenRTM-aist

In this chapter, we will explain how to use the tools provided with OpenRTM-aist, and especially RTCLink, a visual tool to manage RT-Components.

Chapter 5 : The RTM Specification

In this chapter, we will present the RTM Specification, which consists of the collection of interfaces defined within OpenRTM-aist.

Chapter 6 : OpenRTM Class Reference

This chapter contains the reference of all the classes and structures within the OpenRTM-aist framework.

Chapter 7 : OpenRTM IDL Reference

This chapter contains the reference of all the IDL interfaces as defined in the RTM Specification.

Chapter 8 : OpenRTM Extended IDL Reference

This chapter contains the reference of all the IDL interfaces that have been defined within OpenRTM-aist as an extension to the original RTM Specification.

Appendix : How to build the dependencies packages

In this appendix, we will discuss how to install and build the additional packages necessary to compile OpenRTM-aist

- 1- What is the RT-Middleware ?

1.1 Goals of the RT-Middleware

Along with the expansion of the Internet, the number of research and development projects aiming at making robots and robot systems more intelligent by distributing their necessary resources over a network is increasing. Unfortunately, the development of such system usually requires huge investments as well as a large amount of time. These are the main reasons why up until now, we could not witness any success story of a commercial robot reaching a price to functionality ratio high enough to make it widely accepted on the market.

The RT-Middleware aims at establishing a common platform based on the distributed object technology and which would support the construction of various networked robotic systems by the integration of various network enabled robotic elements called RT-Components .

The robotic systems to which we are refering here are not necessarily single bodied robots such as mobile robots or humanoid robots, but more generally speaking "any intelligent networked system using robotic technology and which can perform real world tasks". For example, this also includes systems that although do not look like robots use robotic technology, such as a daily life support or nursing systems in which various sensors and actuators disseminated in the living space collaborate via a networking technology. Actually, any system in which sensing technology, or the fact of processing signals acquired by sensors, collaborates with actuators to provide a feedback into the real world could be assimilated as a robotic system. RT (Robot Technology) is the general term proposed by the Japan Robot Association to designate such a technology.

The RT-Middleware provides a common platform for the RT technology and aims at boosting the efficiency of the research and development in robotics, extending the scope of its applications and fostering the emergence of new markets. Its development is a continuous process. A serious problem often pointed out in the development of software supporting robotic systems compared to the development of traditional software is the lack of reuse due to the specificity of each robotic system. From this observation, the idea emerged of developing a middleware as the common platform to assist in the development of robotic systems in which a large number of users could be involved and which would promote reusability via the modularization of the software involved in the realization of the robotic technology.

1.2 Current state and direction of the RT-Middleware

We, the National Institute of Advanced Industrial Science and Technology - Intelligent Systems Research Institute - Task Intelligence Research Group, have undertaken most of the development of OpenRTM-aist. Initially, the project for the research and development of a

distributed middleware for robotics took place from fiscal year 2002 to 2004 as a part of the 21st century robot challenge program called “Development of the key enabling technologies for Robotics” directed by the New Energy and Industrial Technology Development Organization (NEDO). This research project resulted in several specifications of at the distributed middleware interface level and the delivery of of a prototype implementation. This book is the manual describing the use of the prototype implementation OpenRTM-aist-0.2.0.

During the aforementioned project, discussions regarding the concepts underlying the RT-Middleware and the definition of the scope of the interfaces lead to the implementation of a basic prototype. However, at this stage, we could only provide a sample specification and implementation which is not yet suitable to be used in a real system. From now on, we need to develop real reference implementation supporting more detailed and practical specification based on the the feedback of many actual users. Thank you for understanding that OpenRTM-aist-0.2.0 is,at this stage, only a foothold implementation which still requires extented improvements.

1.3 Why the RT-Middleware ?

Following, we will bring up the strengths of the RT-Middleware.

Support for the specificities of RT-Systems

So far, many middlewares have been proposed to support network robots, intelligent appliances or even computerized houses, and which mainly focus on the IT aspects of such systems. They mostly specialized in

- Sending commands to individual equipments
- Streaming information

While the focus was set on being able to control each individual device over a network, ways to allow collaboration between these devices has seldom been considered. The RT-Middleware was designed to facilitate the collaboration between devices by explicitely differenciating the command flow from the data flow. Components can accomplish tasks using functionalities provided by other components on the network, as if they were its own. Also, the main difference between other general purpose object oriented middlewares like CORBA or DCOM, lies in th fact that the RT-Middleware supports functionalities typical to RT-Systems such as the collaboration between components or the fact that the components have their own activity.

The possibility to develop highly scalable component-based systems

The approach in traditional single bodied robots to develop one monolythic control program made it difficult to integrate various component so as to form a new robotic system. The modularization (hardware and software) of RT related technologies, will not only allow to

create new robotic systems, but also opens the RT technology to applications other than robotics. Moreover, the same RT-Middleware framework can cover a wide range of aspects down to the hardware level. This high degree of scalability also allows for the development and use of components of various granularity that can be finely and freely selected and integrated to adequately convert a wide range of needs.

Platform and network independence

It is hard to provide modularity with a free degree of granularity when being restricted to specific platforms, OS or network. The essence of RTM is not to be bound to any specific architecture so as to be available on any CPU, OS or networking hardware. Basically, RTM can be implemented on any platform supporting CORBA, independently from the OS and programming language used.

Reuse of existing software assets

With the RT-Middleware, software reuse becomes easy thanks to the easy to learn framework and tools provided. Moreover, components produced can be reused not only by their developer but also by independent integrators who be then be able to easily realize new concepts of robotics systems.

Specifications based on international standards

The RT-Middleware makes use of CORBA, a middleware technology for network communication and issued by the OMG (Object Management Group). The RT-Middleware specifications themselves leverage other domain standard issued by the OMG such as the SDO (Super Distributed Object) model. The OMG itself is now recognizing the importance of modularization for the robotic technology and is starting considering how MDA (Model Driven Architecture) can be applied to our standardization activity.

1.4 RT-Middleware and RT-Components

The RT-Middleware, as mentionned before, is a middleware that supports the development of RT related platforms. Figure 1.1 depicts the general outline of the RT-Middleware. The RT-Middleware consists of :

- a component-based framework,
- a set of standard and reusable software modules,
- a set of libraries,
- a set of standard services

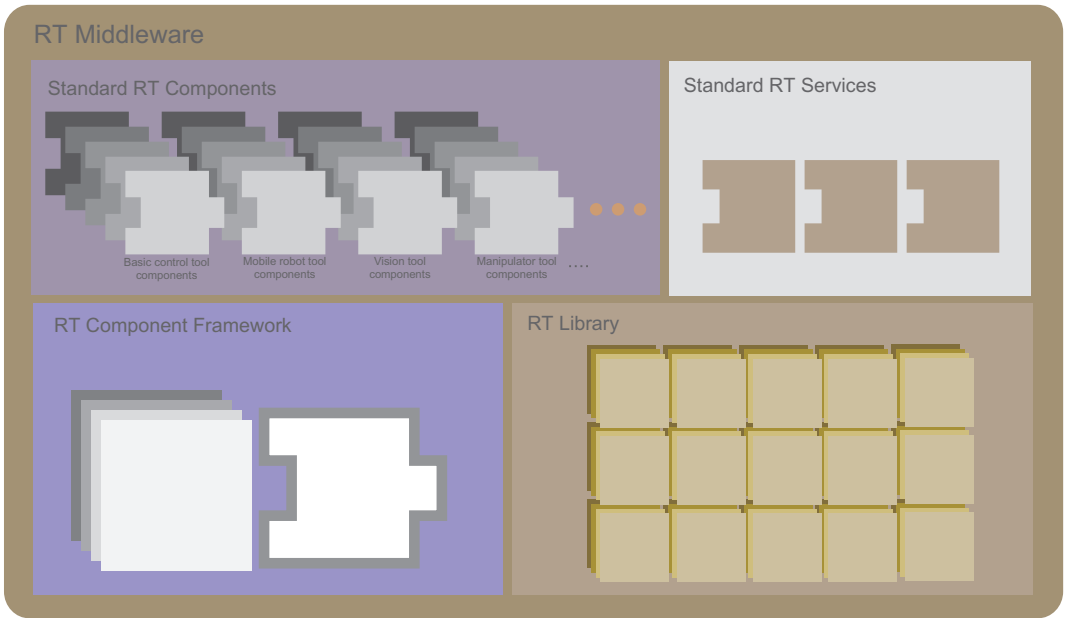

Fig. 1.1: The RT-Middleware

In the RT-Middleware, indepedent software elements are organized into an RT system. These elements are called RT-Components. RT-Components are built using the RT-Component Framework mentioned above. Any system can be designed and realized by combining RT-Components, either by reusing existing or developing your own RT-Components. All the functionalities necessary to manage systems of RT-Components are provided by a set of services, independently from the component themselves. The RT-Middleware also provides library to assist in the development of RT-Components.

The technology supporting all the above mentioned functionality is what we call the RT-Middleware.

1.5 RT-Components

We will now explain the basic structure of an RT-Component. Figure 1.2 represents the architecture block diagram of an RT-Component. RT-Components are designed and implemented using the distributed object technology so that functionalities provided by diffrent component over a network can be accessed transparently.

One of the main considerations for OpenRTM-aist being the OS and programming language independency, it has been implemented on top of CORBA.

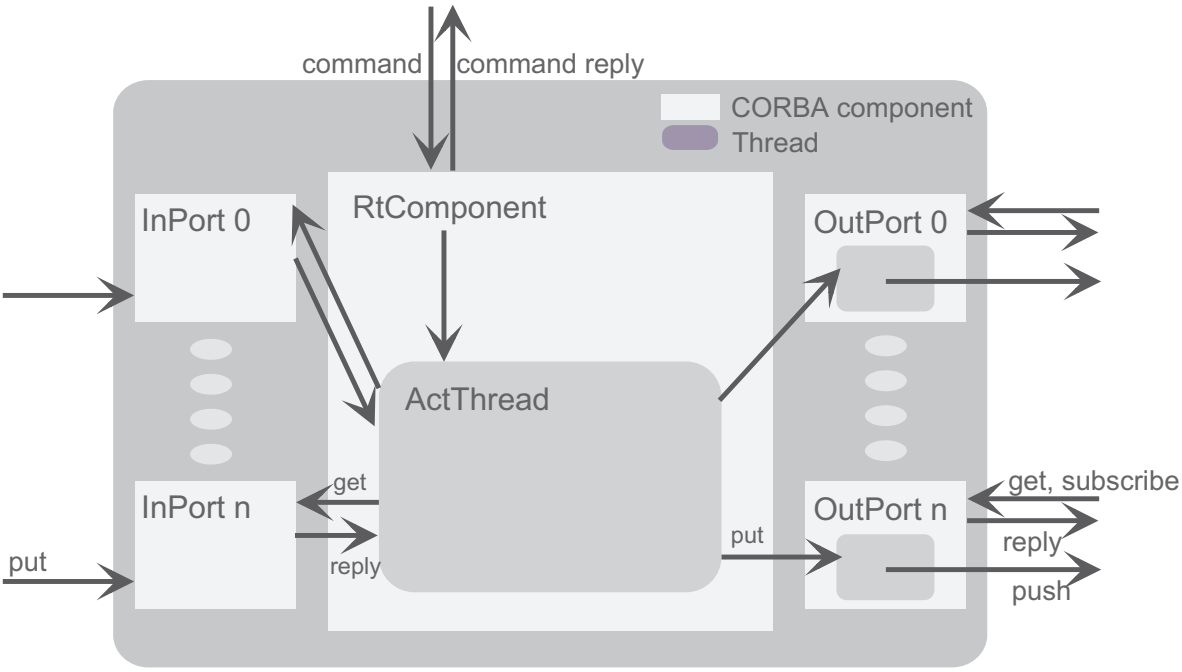

Fig. 1.2: RT-Component

In a traditional distributed object, the exchange of data is basically realized via method calls. However, in the input and output streaming of data mechanism involved in robot control, the type data exchanged is more important than the method itself.

For example, considering a joystick with 6 degrees of freedom, the 6 double typed values it provides can be used to control different objects (Manipulator, Mobile Robot, Humanoid Robot,etc..) in different ways (control the position, control the speed, control the impedance, etc...). In a system built by only applying the traditional principles of object distribution, the joystick component would have to know in advance (at design time) all the methods (interfaces) supporting the different types of control. However, as long as the sender is able to send these 6 double typed values, it should not have to know how the receiver will process this data, as long as the receiver is able to accept these 6 double typed values. This is why an RT-Components owns directional data ports called InPort/OutPort, which allows the exchange of data between component and that can be connected as long as the type of data they provide is identical.

An RT-Component is composed of several objects which can be roughly classified into the 3 following categories :

RTComponent This is the main body of the RT Component which provides the basic interfaces and provides an access to its 0∼n InPorts / OutPorts. It supports 1 core logic process and its internal state will vary in response to external or internal events. This core logic process, called the “Activity”, is executed in an independent thread and is therefore decoupled from the data exchange mechanism.

InPort This is the input port that handles the data received from other components. A component that needs to receive streams of data should do it by using InPorts. As the core logic will process data received by the InPorts sequentially, InPort is particularly well suited to periodic components.

OutPort This is the output port that will stream the processed data to other components. It supports both the pull mode, in which the receiver acquires the data by itself, and the push mode, in which the data is actively sent receivers that subscribed.

OpenRTM-aist provides the necessary framework and tools that facilitate the building of such RT-Components. Using them will allow you to wrap any legacy software (like robot control libraries, etc...) into an RT-Component. Moreover, software that has been made into components can be combined together into a system, without beeing recompiled as long as its specification remains unaltered.

1.6 RT-Conponents, Manager and Naming Service

As described above, RT-Components provide various functionalities that allow the development of robotic systems. However, such system cannot be realized only with components. It is also necessary to be able to start, stop, interconnect and discover components that are deployed on several host.

OpenRTM-aist provides with a component manager, an object that will be in charge of managing the lifecycle of the components, register them to the naming service, etc...

The relation between a component, its manager and the naming service is depicted in Figure 1.3.

Usually, a component is provided as a loadable library (or module), which the manager will load and instantiate in order to create a component.

Then, in order to make this component visible to other components on the network, the manager will, using the CORBA naming service, register the name of the component as well as its object reference to a name server. Finally, the component will also have the possibility to discover other components by querying the name server through its manager.

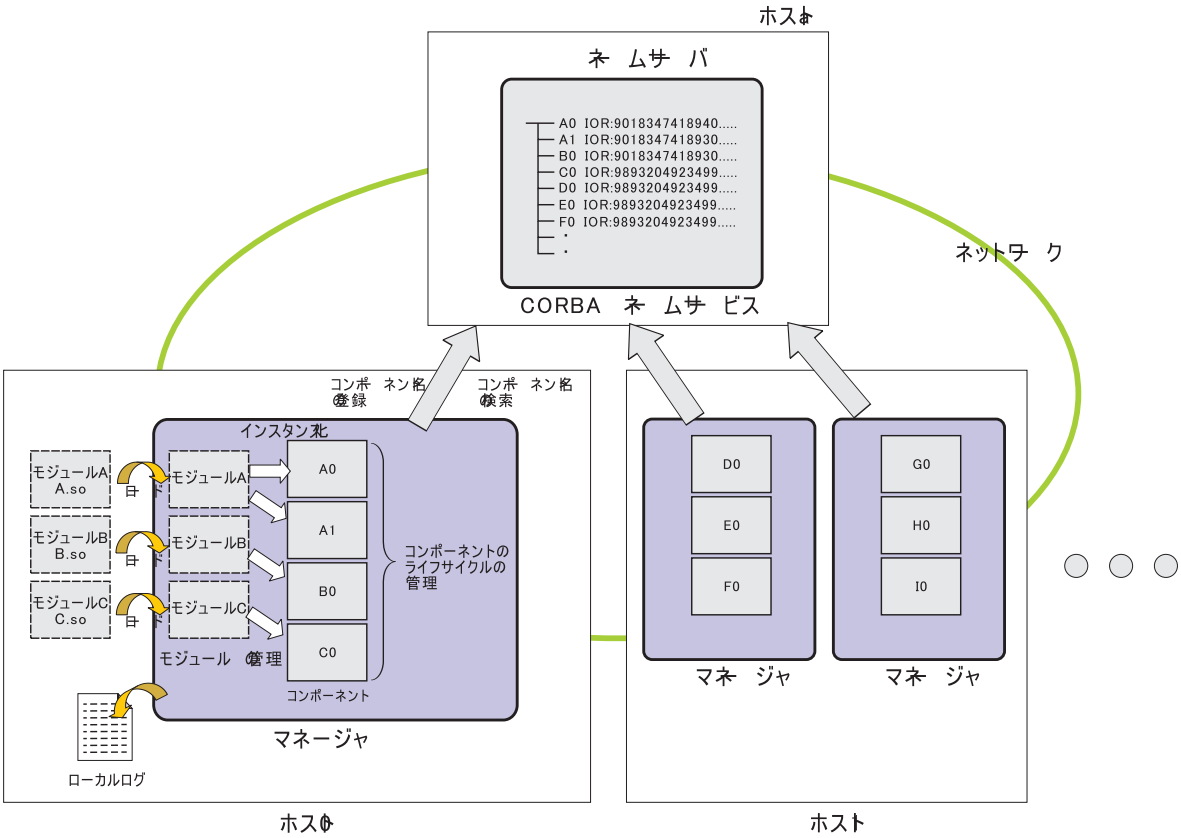

Fig. 1.3: RT-Component, Manager and Naming Service

- 2- Introduction to RT-Components

2.1 Installing OpenRTM-aist

In this chapter, we will explain on how to build OpenRTM-aist from the source code and install it. If you are planning to build and install OpenRTM-aist form its source, follow the procedure described below. If you are only planning to use and link to the binary packages, feel free to skip this chapter.

2.2 Building and installing OpenRTM-aist in a UNIX environment

2.2.1 Prerequisites

The following packages are necessary when installing OpenRTM-aist. A basic explanation on how to instal these packages is provided in the appendix at the end of this book. For more details on the installation procedures, refer to the documentation provided with each package.

● **List of dependency packages**

ACE	The ADAPTIVE Communication Environment (ACE) 5.4 or later URL: `http://www.cs.wustl.edu/~schmidt/ACE.html`
boost	boost C++ library 1.30 or later URL: `http://www.boost.org/`
ORB	As of today, the only ORB supported is omniORB-4.0 or later omniORB – URL: `http://omniorb.sourceforge.net/`
Python	Python 2.2 or later. In order to develop components in Python, a Python development environment is necessary. URL: `http://www.python.org/`
gcc	gcc version3 or later is prefered. Although it is possible to compile with gcc2.95.x, some tweaking is necessary (see below) URL: `http://gcc.gnu.org/`

In order to make the compilation of OpenRTM-aist easier, we recommend that you install all these packages in the default directories (/usr,/usr/local...). The default directory for command files, header files and libraries is as indicated below.

● **Default installation directories**

Command files	`/usr/bin, /usr/local/bin`
Header files	`/usr/include/`*package_name*, `/usr/local/include/`*package_name*
Libraries	`/usr/lib`, `/usr/local/lib`

In the recent Linux distributions, the default packages are installed under /usr/ in [bin—include—lib]. On other UNIX-like OSes, they can usually be found under /usr/local/ or /opt/ in [bin—include—lib]. As autoconf, automake are used to OpenRTM-aist, we do not recommend, while it is still possible, to install the dependency packages in directories different from the one indicates above.

In order to avoid later trouble, we strongly recommend that you install all the packages in the defeult directories.

● **Dependency packages target platforms**

debian GNU Linux	`http://www.debian.org/distrib/packages`
	ace – ○, boost – ○, omniORB – ○, Python – ○
RedHat/Fedora-like	`http://rpmfind.net/`
	ace – ○, boost – ○, omniORB – ○, Python – ○
Vine Linux	`http://vinelinux.org/`
	ace – ×, boost – ×, omniORB – ×, Python – ○
FreeBSD	`http://www.freebsd.org`
	ace – ○, boost – ○, omniORB – ○, Python – ○

AIST provides versions of ace, boost and omniORB (+ omniORBpy) adapted to Vine Linux. Refer to the OpenRTM-aist homepage (http://www.is.aist.go.jp/rt/).

2.2.2 Decompressing the Source files

The first task is to decompress the file OpenRTM-aist-0.2.0.tar.gz into a suitable directory. It could be either one's own home directory or any other directory.

However, you should select the directory so as to be able to perform the following operations as a general **user** rather than **root**.

```
> tar xvzf OpenRTM-aist-0.2.0.tar.gz
> cd OpenRTM-aist-0.2.0
```

2.2.3 Building OpenRTM-aist

The building of OpenRTM-aist is performed using autoconf, automake, in the same way as any other package using autoconf, automake. It is however necessary to first set the options related to the ORB.

```
> ./configure [options]
```

Below is a partial list of options that can be passed to `./configure`.

2.2.3.1 Setting the ORB

※ Presently, only omniORB is supported. We plan to support other options in the future. It is possible to specify the directory where the ORB has been installed using the additional parameter `=dir`.

● ORB selection option

`--with-omniorb=dir`	The ORB is omniORB.
`--with-tao=dir`	The ORB is TAO.
`--with-mico=dir`	The ORB is MICO.
`--with-orbix=dir`	The ORB is Orbix/E.
`--with-orbacus=dir`	The ORB is ORBacus.

2.2.3.2 Setting the programming language

C++ is the prefered and default programming language for the development of components using OpenRTM-aist. Presently, the only other supported language is the scripting language Python (version 2.2 or later). In order to develop components in Python, it is necessary to set the following option. If this option is set, a Python development environment must be available on your system.

If Python has been installed elsewher than the default directory, you must specify the installation directory using `=dir`. For example, if the files `Python.h` resides in the directory *python_dir*`/python2.x/Python.h`, the setting should be `--with-python=`*python_dir* .

● **Programming language selection options**

`--with-python=dir`	Install the Python interfaces.

2.2.3.3 Compiling using a version of gcc prior to version 3.x

Using gcc prior to version 3, some reported compilation errors when trying to instantiate a template class that was too deep. To fix this problem, the compilation option `-ftemplate-depth-n|` must be set. Therefore, when using gcc-2.95.x, passing the following option to `./configure` will automatically set this complation option.

● **Compiler selection option**

`--with-gcc2`	The compiler is gcc version 2.x

2.2.3.4 Path to the ACE include and library files

If ACE was not installed in its default directory, the following option will allow to set the path to its header and library files.

● **ACE header/library files directory option**

`--with-ace-includes=dir`	Specifies the directory for the ACE header files
`--with-ace-lib=dir`	Specifies the directory for the ACE library files

When decompressing the ACE archive, the source files are copied in a directory called `ACE_wrappers`. The header and library files will be decompressed in the sub-directory called `ACE_wrappers/ace`. If the archive was decompressed so that `ACE_wrappers` is actually `/tmp/src/ACE_wrappers`, the following option should be set.

```
./configure ... --with-ace-include=/tmp/src/ACE_wrappers \
                --with-ace-libs=/tmp/src/ACE_wrappers/ace ...
```

Note that as in ACE, all header files are refered to assuming that they have been placed in the sub-directory `ace` (`#include "ace/ACE.h"`, the path to the header files should actually be set to `/tmp/src/ACE_wrappers`. On the other hand, as libraries are refered to directly in the directory they where placed, the setting should be `/tmp/src/ACE_wrappers/ace`.

2.2.3.5 Other options

The option `-help` displays a list of all available options. Refer to it for options that were not described here.

```
> ./configure --help
```

2.2.4 configure, make

After setting the options adapted to you system environment, execute `configure` as folow. `configure` will then generate the necessary Makefiles, header files, dependency reference files and display the compilation options. Make sure `configure` executed properly.

```
> ./configure --with-omniorb --with-python
        :
        :
------------------------------------------------------------

  OpenRTM-aist will be compiled with the following environment.

------------------------------------------------------------
CXX: g++
CPPFLAGS: -I/usr/local/include -I/usr/local/include \
          -I/usr/local/include -I/usr/local/include
CXXFLAGS: -I/usr/local/include -O
BUILD_CFLAGS:
BUILD_CPPFLAGS:

LD: /usr/bin/ld
LIBS: -lpthread -lACE -lboost_regex -lomniORB4 -lomnithread -lomniDynamic4
LDFLAGS:  -L/usr/local/lib -L/usr/local/lib -L/usr/local/lib
LDSOLIBS:  -lACE -lboost_regex -lomniORB4 -lomnithread -lomniDynamic4

ORB: omniORB
IDLC: /usr/local/bin/../bin/omniidl
IDL_FLAGS: -bcxx -Wba -nf

WRAPPERS:
------------------------------------------------------------

configure: creating ./config.status
config.status: creating Makefile
        :
        :
config.status: creating rtm/config_rtc.h
config.status: executing depfiles commands
```

If the last line displayed is not `config.status: executing depfiles commands`, it means `configure` did not execute properly. In that case, check the `config.log` file to identify the cause of possible errors or a warnings. Usually, the cause is that the all necessary dependency packages have been properly installed or that the header and library files cannot be found. You should check twice the pathes provided in the options.

If `configure` executed properly, we can now run `make`. The building process can take from a few minutes up to 20 minutes.

```
> make
```

All directories are built recursively. In order to complete the installation, at least the following directory should be built successfully.

● Organization of the folders in OpenRTM-aist

`OpenRTM-aist-0.2.0/rtm`	RT-Component Framework
`OpenRTM-aist-0.2.0/rtm/idl`	RT-Component IDLs
`OpenRTM-aist-0.2.0/util/rtm-config`	RTM Configuration Tool
`OpenRTM-aist-0.2.0/util/rtm-naming`	RTM Naming Service Wrapper
`OpenRTM-aist-0.2.0/util/rtc-template`	RTC Template Tool
`OpenRTM-aist-0.2.0/util/rtc-link`	Component Management GUI

After the build has completed successfuly, we now have to install the header files, library files, utilities and documentation. To install these files in a system directory (like `/usr` or `/usr/local`), the root privileges are necessary.

```
> su
# make install
```

After becoming root, running `make install` will install the header and library files in the system directory.

2.2.5 Testing the sample programs

Once the installation completed, you can test it on the sample application example/SimpleIO. With this simple application, a component will send numerical values entered via the console to another component that will directly display them in its console.

```
> cd examples/SimpleIO/
> run.sh
```

From here, we will assume that executing `run.sh` will open a kterm terminal window. If you wish not to use kterm but another terminal window, you can do so by modifying the `run.sh` shell script accordingly.

At first, 2 kterm windows will appear, then after a few seconds, another one will appear. You must wait until the third window appears. You can then type numbers in the window called ConsoleIn. These numbers sould fit in a `long int` structure. The numbers typed in will simultaneously be displayed in the console called ConsoleOut, along with the time (the format for the time is sec:nsec). Reaching this stage indicates that the build and installation of the `libRTC.so` library was successful.

2.2.6 Testing the component template

Next, we will test the component template generator by writing a simple component. We will use the `rtc-template` template generator. After making the directory where we will generate the component (any directory will do), we will generate and compile the component.

```
> mkdir SampleComponent
> cd SampleComponent
```

Let's first take a look at the help provided by `rtc-template`.

```
> rtc-template --help

Usage: rtc-template [OPTIONS]
Options:
    [--help]                              Print this help.
    [--c++]                               Generate C++ template code.
    [--python]                            Generate Python tempalte code.
    [--output[=output\_file]]              Output base file name.
    [--module-name[=name]]                Your module name.
    [--module-desc[=description]]         Module description.
    [--module-version[=version]]          Module version.
    [--module-author[=author]]            Module author.
    [--module-category[=category]]        Module category.
    [--module-comp-type[=component\_type]] Component type.
    [--module-act-type[=activity\_type]]   Component's activity type.
    [--module-max-inst[=max\_instance]]    Number of maximum instance.
    [--module-lang[=language]]            Language.
    [--module-inport[=PortName:Type]]     InPort's name and tyoe.
    [--module-outport[=PortName:Type]]    OutPort's name and type
       :
       :
Example:
    rtc-template --c++ --module-name=Sample --module-desc='Sample component' \
    --module-version=0.1 --module-author=DrSample --module-category=Generic \
    --module-comp-type=COMMUTATIVE --module-act-type=SPORADIC \
    --module-max-inst=10  \
    --module-inport=Ref:TimedFloat --module-inport=Sens:TimedFloat \
    --module-outport=Ctrl:TimedDouble --module-outport=Monitor:TimedShort
```

We will generate the skeleton of a component by passing its specifications as arguments to `rtc-template`. Here, we will try the "Example" provided at the end of the help section.

```
> rtc-template --c++ --module-name=Sample --module-desc='Sample component' \
    --module-version=0.1 --module-author=DrSample --module-category=Generic \
    --module-comp-type=COMMUTATIVE --module-act-type=SPORADIC \
    --module-max-inst=10  \
    --module-inport=Ref:TimedFloat --module-inport=Sens:TimedFloat \
    --module-outport=Ctrl:TimedDouble --module-outport=Monitor:TimedShort
Sample.h was generated.
Sample.cpp was generated.
SampleComp.cpp was generated.
Makefile.Sample was generated.
> ls
Makefile.Sample         Sample.h
Sample.cpp              SampleComp.cpp
```

The C++ source files and the Makefile `Makefile.Sample` have then be generated. Using the generated Makefile, you can now try to compile the source files.

```
> make -f Makefile.Sample
```

or

```
> mv Makefile.Sample Makefile
> make
```

As indicated, you can either directly use the `Makefile.Sample` by passing it to `make` along with the `-f` option, or rename `Makefile.Sample` to `Makefile` and simply call make.

```
> make -f Makefile.Sample
rm -f Sample.o
g++ ‘rtm-config --cflags‘ -c -o Sample.o Sample.cpp
     :
     :
g++ ‘rtm-config --libs‘ -o SampleComp Sample.o SampleComp.o
rm -f Sample.so
g++ -shared ‘rtm-config --libs‘ -o Sample.so Sample.o
> ls
Makefile.Sample  Sample.h  Sample.so*   SampleComp.cpp
Sample.cpp       Sample.o  SampleComp*  SampleComp.o
```

Doing so, both the loadable module version (`Sample.so`) and the executable version (`SampleComp`) of the component were built. Let's try to run the executable version of the component. A configuration file (usually named `rtc.conf`) must be provided when executing a component. Here, we will create a very simple one in the current directory.

```
> cat > rtc.conf
NameServer      Current_Host_Name:Port_Number
^D (Ctrl+D)
```

In this example, we will set the hostname to rtm.or.jp and the port number to 6789. You should replace rtm.or.jp by the actual hostname of your machine.

```
> cat > rtc.conf
NameServer      rtm.or.jp:6789
^D (Ctrl+D)
```

We can confirm by typing:

```
> cat rtc.conf
NameServer      rtm.or.jp:6789
```

Next, we start the CORBA Naming Service. The CORBA Naming Service can be started by inputing the following command :

`rtm-naming Port Number`

The port number passed as argument should be the same as the one defined in rtc.conf.

```
> rtm-naming 6789
Starting omniORB omniNames: ichi:9999
n-ando@ichi:/tmp/SampleComponent>
Fri Oct 29 17:12:51 2004:

Starting omniNames for the first time.
Wrote initial log file.
Read log file successfully.
Root context is IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f
4e616d696e67436f6e746578744578743a312e3000000100000000000000600000000010102000e00
00003135302e32392e39362e313638000f270b0000004e616d6553657276696365500020000000000
0000080000000100000000545441010000001c0000000100000001000100010000000100010509010
01000100000009010100
Checkpointing Phase 1: Prepare.
Checkpointing Phase 2: Commit.
Checkpointing completed.
```

Next, we will start the component.

```
> SampleComp -f rtc.conf
```

The component will then start. If you could reach this stage, it is safe to assume that OpenRTM-aist was installed correctly.

2.3 Debugging using RTCLink

In our previous SimpleIO sample program, the two components were started and their connection established by a third program. However, at the development stage, testing various combinations of components using dedicated programs can soon become really cumbersome and does not really take advantage of RT-Component concept. OpenRTM-aist therefore provides a graphical tool (RTCLink) to help with starting and connecting components.

We will now show how to connect the aforementioned ConsoleIn and ConloseOut components using RTCLink. For more details on how to use RTCLink, refer to the following section called [4.1 RTCLink].

2.3.1 How to do it

Let's first start RTCLink. You can do so by rtc-link in the console window.

In the directory where SimpleIO resides, we then must start `ConsoleInComp` and `ConsoleOutComp` in two different windows. (See figure 2.1 and 2.2).

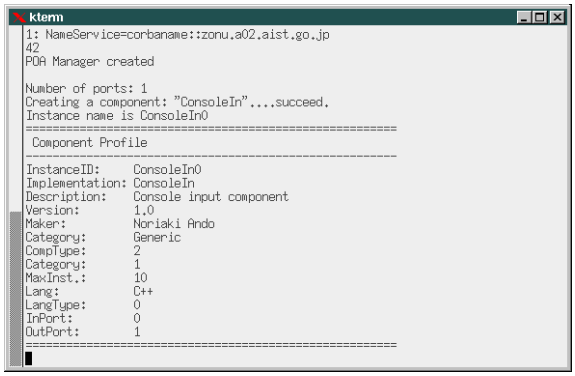

Fig. 2.1: Starting `ConsoleInComp`

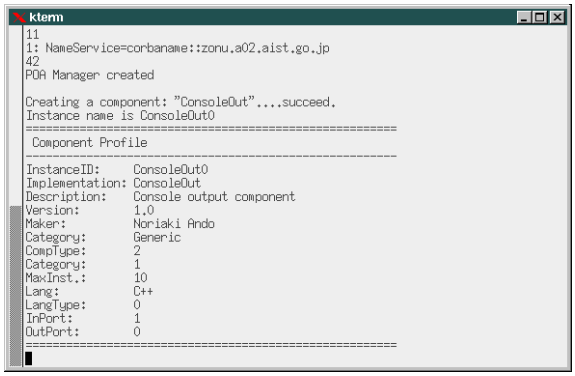

Fig. 2.2: Starting `ConsoleOutComp`

Next, in the Tree Pane at the left hand side of the RTCLink window, we will expand the tree by clicking on it. Make sure that the ConsoleIn and ConsoleOut components are displayed in the tree (Figure 2.3). The component is displayed using the following icon :  .

Moreover, by clicking the  button in the toolbar, the System Composer Pane appears in the central part of the window (Figure 2.4). We will compose the system in this window.

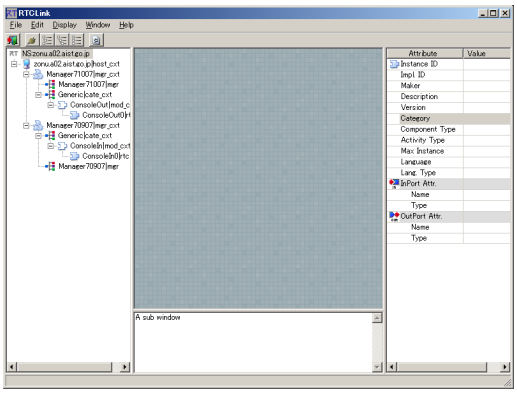

Fig. 2.3: Expanding the Tree

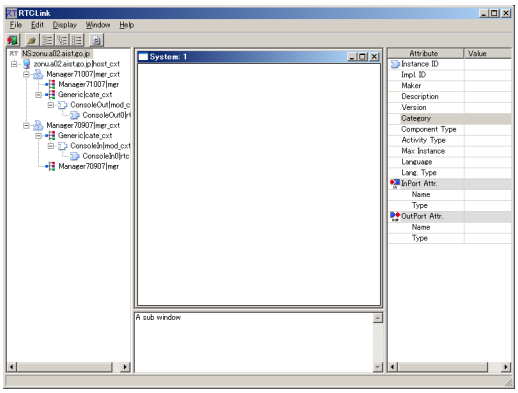

Fig. 2.4: Displaying the system Composer Window

We will now drag and drop the ConsoleIn and ConsoleOut components represented by the  icon from the tree pane to the system composer pane.

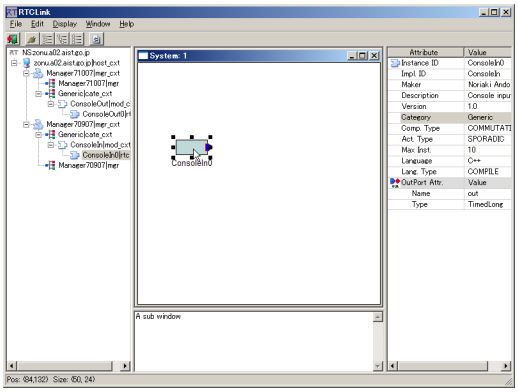

Fig. 2.5: Placing the ConsoleIn component

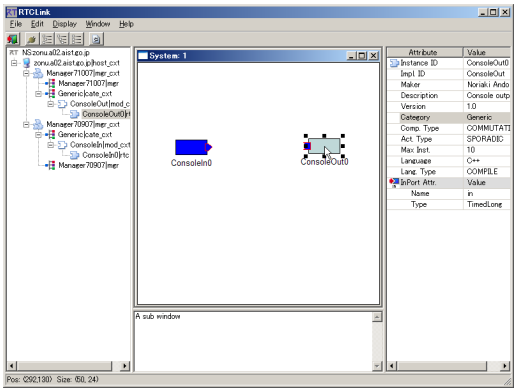

Fig. 2.6: Placing the ConsoleOut component

As indicated in Figure 2.7, we will then click the OutPort of the ConsoleOut component and drag it onto the InPort of the ConsoleIn component. Doing so will connect the OutPort on the ConsoleOut component to the InPort on the ConsoleIn component.

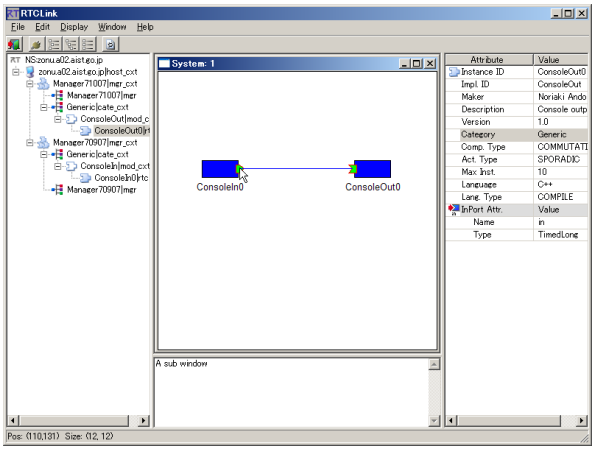

Fig. 2.7: Connecting an OutPort to an Inport

Next, by a right-click on the ConsoleIn component in the system composer pane, we will make the context menu appear. The ConsoleIn component will be started upon selection of the “Start” item in the menu (Figure 2.8). In the same way, we will start the ConsoleOut component (Figure 2.9).

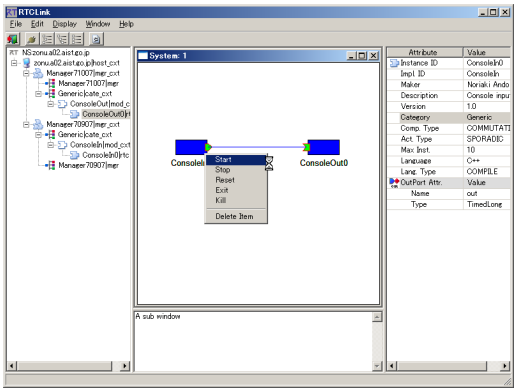

Fig. 2.8: Starting ConsoleIn

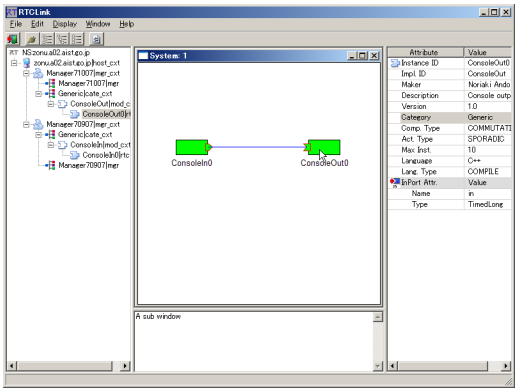

Fig. 2.9: Starting ConsoleOut

After being started, ConsoleIn will start prompting for numerical inputs (Figure 2.10. You can then type numbers that sould fit in a `long int` structure. The numbers typed in will simultaneously be sent to the ConsoleOut component.

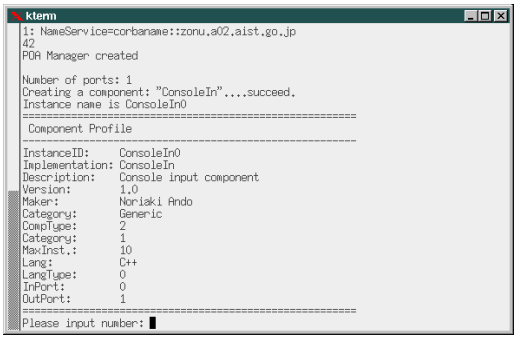

Fig. 2.10: ConsoleIn window after starting

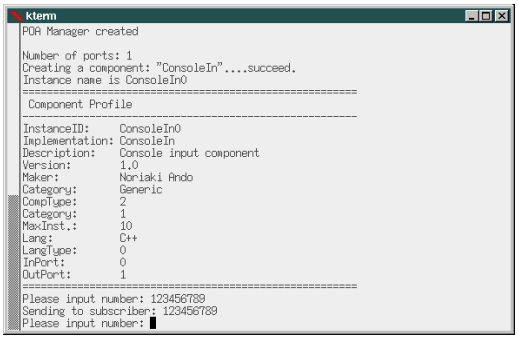

Fig. 2.11: Inputing numbers in ConsoleIn

The ConsoleOut component, after starting, will output numerical values along with the time (Figure 2.12). However, if there is no input from ConsoleIn, it will continuously output 0. When inputing a number into ConsoleIn, this number will be displayed as in figure 2.13.

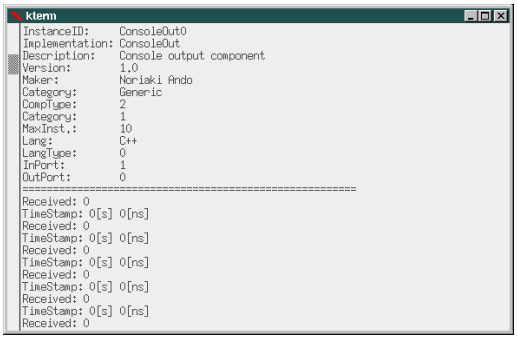

Fig. 2.12: ConsoleOut window after starting

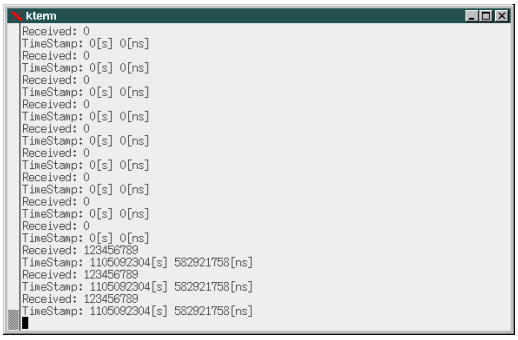

Fig. 2.13: Numbers displayed in ConsoleOut

If this sample runs properly, we can assume that OpenRTM-aist is working properly.

2.4 Building and installing OpenRTM-aist in a Windows environment

Support for the Windows platform is planned for a future version of OpenRTM-aist.

- 3- Programming an RT-Component

3.1 Development flow of an RT-Component

OpenRTM-aist provides a framework that enables its users (the component developers) to easily adapt existing or develop new software that complies with the RT-Component specifications.

The general development flow of an RT-Component is described in figure 3.1.

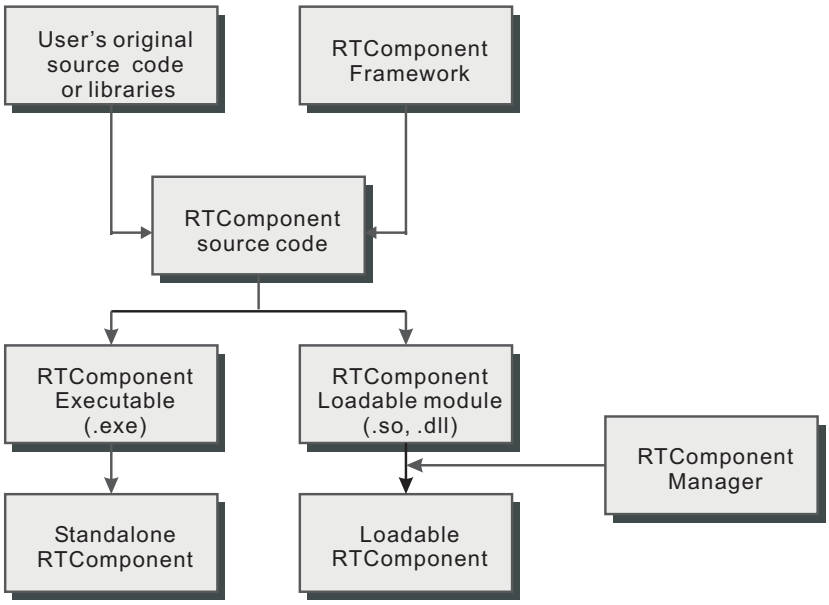

Fig. 3.1: RT-Component development flow

The component developer can create new components by integrating existing software libraries into the component framework. By doing so, the functionalities offered by legacy software will be modularized as RT-Components and will become reusable through different ranges of applications. The newly so developed RT-Component can then be deployed on any appropriate network node but will become, by supporting the distributed object technology, usable from any location on the network.

As described on figure 3.1, basically two forms of executable binary files can be generated from an RT-Component developed in conformance with th RT-Component framework. A standalone RT-Component is a binary file that can be directly executed as is. A loadable module RT-Component is a binary file that can be dynamically loaded into an application. An RT-Component can support both these two patterns of development, deployement and execution.

3.1.1 Morphology of an RT-Component

As previously stated, an RT-Component can be built either as a Standalone RT-Component, which can be executed as is, or a Loadable Module RT-Component, which can be dynamically loaded into an application at run-time. Both forms of an RT-Component, Standalone RT-Component and Loadable Module RT-Component, will be described below.

3.1.1.1 The Standalone Component

This form of binary allows the component to be executed as an independent entity, As a general rule, using this form, only one type of component can run in a single process (although several instances of the same component are possible). As all communications with other components occur via the CORBA stack, there may be some performance concerns. However, a fatal error occuring in another component will not affect a standalone component, which makes it a safer.

Also, the starting and stopping of a component coincides with the strating and the stopping of the process itself. Therefore, it can be done without the risk of affecting other components.

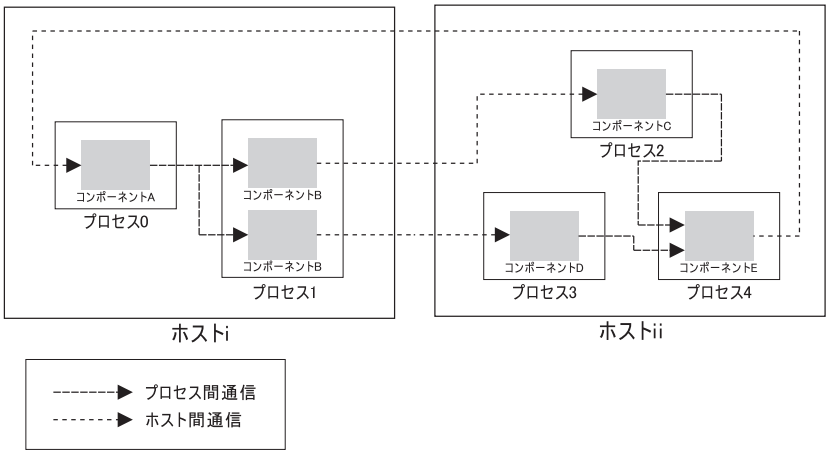

Fig. 3.2: Standalone Component

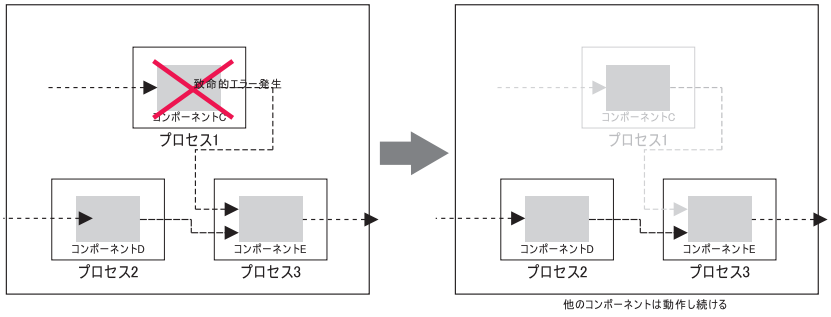

Fig. 3.3: Influence of an error on a standalone component

3.1.1.2 Loadable Module Component

This form of component is dynamically loaded and instanciated by a "Component Manager".

The Loadable Module Component RT-Component is usually a shared library. A shared library is a library is dynamically loaded at runtime. In a UNIX environment, its file suffix is `.so` (`.sl` for HP-UX), while in a Windows environment, it is `.dll`. As a dynamically loadable component does not support the `main` function, it can not be executed by itself, To execute a dynamically loadable component, a component server first has to load load it and request the internal component manager to instantiate and execute it.

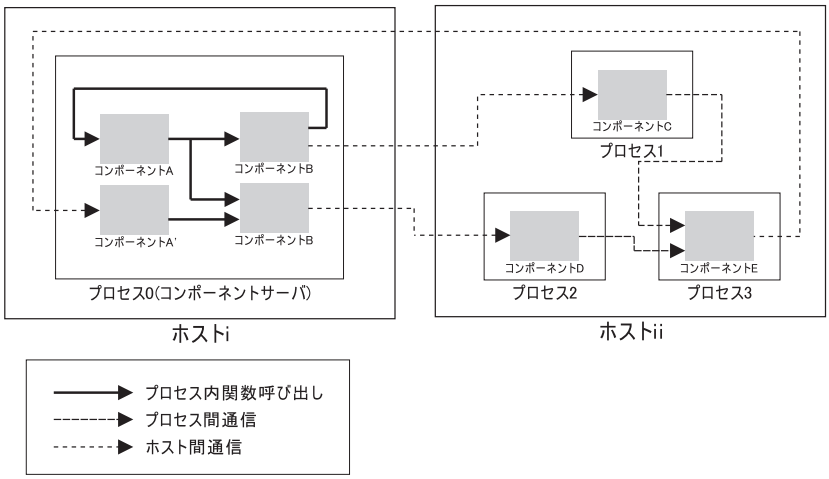

Fig. 3.4: Loadable Module Component

In a loadable module component, a single process can manage several instances of several types of components (Figure 3.4). Most CORBA implementations support optimization mechanisms to speed up calls to operations on CORBA objects residing in the same process. Therefore, by instantiating and connecting two highly inter-dependent components within the same component server, we can expect significant performance improvements.

It is however to be noted that, as it greatly depends on the CORBA implementation, it is not possible to draw a general rule regarding to the possible performance gain. Therefore, we can only speculate that, when developing a system with stringent speed requirements, using loadable module components should lead to some amount of performance gain.

The loadable module component and the standalone component mainly differ in the fact that if one of the component running within the process causes a segment violation fatal error and crashes, the host process, with all other components running within it will crash as well (Figure 3.5). Therefore, a component that has not been properly tested let to execute along with other safer components may hinder the execution of the later and geopardized the safety of the whole system.

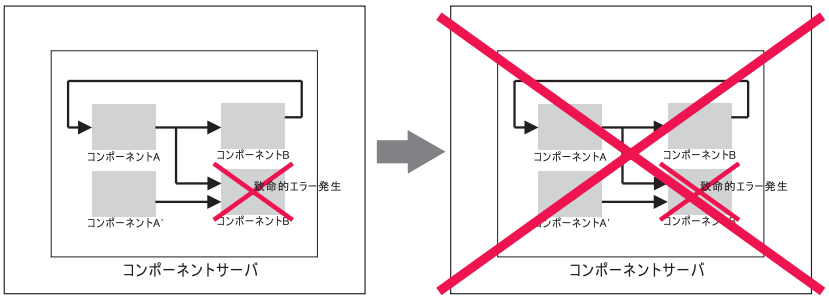

Fig. 3.5: Influence of an error caused by a loadable module component

Once its interface has been defined, it has been thoroughly tested and has reached an acceptable level of performance, a component can be used without its user ever to have to deal with its source code. It can simply be assembled with other components to form a system. This is one of the big selling point for using RT-Components. The component developer is free to distribute either the source code of his component, its compiled loadable module version or its executable version, or all of them.

Module versus Component

In OpenRTM-aist, the words "Module" and "Component" have two distinct signification as described below.

● **Module versus Component**

Module	Class that defines the model supported by the component
Component	Embodiement of the module class. Called RT-Component, it is what actually forms a system.

From the C++ language perspective, a module would be equivalent to a class, while a component would be an instance. A component is the realization of a module. Therefore, from a module A, several components (A0,A1,..., An) can be instantiated (Figure 3.6).

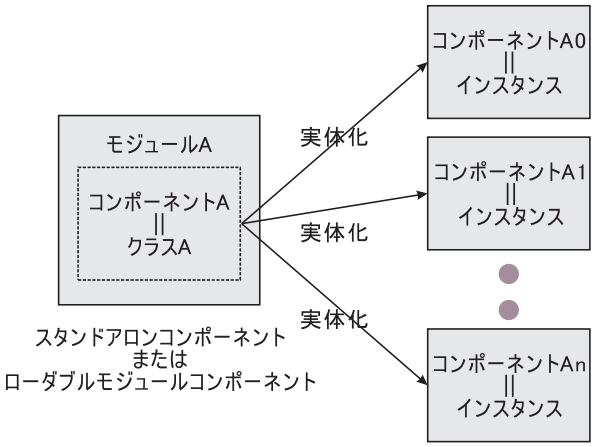

Fig. 3.6: Module vs. Component

While the component is the entity which does the actual processing, a module could be considered as its kind of template. OpenRTM-aist provides the framework containing the base classes used to create such templates. The component developer creates his own components by deriving these base classes.

3.2 Development of a Component

From now, we will explain how to actually create components by following a simple example. The source files for the sample presented below is available in the directory called `example`.

3.3 The "Hello RT World" Sample

Let's build our first "Hello RT World" component.

The "Hello RT World" component, once started, will open a console window and output the text "Hello RT World!!!" continuously. This easy component will not do anything else. At this point, we will explain the fundamentals of the component framework.

The following 4 files are necessary to build this component.

● **Files necessary to build the HelloRTWorld Component**

`HelloRTWorld.h`	HelloRTWorld component header file.
`HelloRTWorld.cpp`	HelloRTWorld component source file.
`HelloRTWorldComp.cpp`	HelloRTWorld component executor source file (standalone component).
`Makefile`	Makefile to build the component.

We will define and implement the main class of our component, called HelloRTWorld, in the files `HelloRTWorld.h` and `HelloRTWorld.cpp` respectively. As the class HelloRTWorld actually implements our component, we will make it derive from the `RtcBase` class, which already implements all the basic functionalities that an RT-Component must support.

As we decided to make the HelloRTWorld component standalone, we need to write the code that will instantiate and execute it in the file called `HelloRTWorldComp.cpp`.

At the same time, we will also build the loadable module form of the component. The building steps are described in the `Makefile` file.

3.3.1 Header File

We first define the header of the class. As previously mentionned, we define the class `HelloRTWorld` by deriving the `RtcBase` base class. Depending on your application, the class

`HelloRTWorld` could inherit from several other pre-defined classes. Let's now take a look at the `HelloRTWorld.h` file.

HelloRTWorld.h

```
#ifndef __HELLORTWORLD_h__
#define __HELLORTWORLD_h__


#include <rtm/RtcBase.h>
#include <rtm/RtcManager.h>
#include <rtm/RtcInPort.h>
#include <rtm/RtcOutPort.h>

using namespace RTM;

static RtcModuleProfSpec hellortworld_spec[] =
  {

  {RTC_MODULE_NAME, "HelloRTWorld"},
  {RTC_MODULE_DESC, "Hello RT world component"},
  {RTC_MODULE_VERSION, "0.1"},
  {RTC_MODULE_AUTHOR, "DrSample"},
  {RTC_MODULE_CATEGORY, "Generic"},
  {RTC_MODULE_COMP_TYPE, "COMMUTATIVE"},
  {RTC_MODULE_ACT_TYPE, "SPORADIC"},
  {RTC_MODULE_MAX_INST, "10"},
  {RTC_MODULE_LANG, "C++"},
  {RTC_MODULE_LANG_TYPE, "COMPILE"},
  {RTC_MODULE_SPEC_END, NULL}
  };


class HelloRTWorld
  : public RTM::RtcBase
{
 public:
  HelloRTWorld(RtcManager* manager);
  virtual RtmRes rtc_active_do();
};


extern "C" {
  RtcBase* HelloRTWorldNew(RtcManager* manager);
  void HelloRTWorldDelete(RtcBase* p);
  void HelloRTWorldInit(RtcManager* manager);
};
#endif // __HELLORTWORLD_h__
```

You will find below the description of the code.

- Include guard
- Inclusion of all necessary headers
- Declaration of the namespaces used
- Definition of the Module Profile List
- Declaration of the class HelloRTWorld
- Declaration of the factory and initialization functions

First, the include guard is a predefined statement that will prevent the double inclusion of this header file. Be careful not to forget its counterpart `#endif` at the end of the file.

We then included the headers related to RTM. A component developer must make his component inherit from the `RtcBase` class (or one of its derived class). Therefore, it is necessary to include the file `RtcBase.h` (or the header file in which one of its derived class is defined). Also, as the RT-Component constructor needs a pointer to an instance of `RtcManager` , it is necessary to either include the file `RtcManager.h` or declare the class `RtcManager` .

All classes related to RTM are defined in the namespace called `RTM` . Therefore in order to define the inheritance from `RTM::RtcBase`, as for all classes and operation defined in RTM, it is necessary to either make use of the scope resolution operator `RTM::` or declare the `using namespace RTM` directive.

Next, we declare the Module Profile List. To each module is associated the information such as the name of the module, its developer's name, its version... See below for a complete list of the information stored in the Module Profile List.

●コンポーネントプロファイル

`RTC_MODULE_NAME`	Module Name. Specifies the name of the module (HelloRTWorld)
`RTC_MODULE_DESC`	Module Description. Holds a short description of the module (Hello RT world component)
`RTC_MODULE_VERSION`	Module Version. Specifies the version of the module(0.1)
`RTC_MODULE_AUTHOR`	Module Author. Specifies the name of the module developer (DrSample@AIST, Japan)
`RTC_MODULE_CATEGORY`	Module Category. Specifies to which category belongs the module (Generic)
`RTC_MODULE_COMP_TYPE`	Component Type. Specifies which type of component will realize the module (COMMUTATIVE)
`RTC_MODULE_ACT_TYPE`	Activity Type. Specifies the type of activity supported by the module (SPORADIC)
`RTC_MODULE_MAX_INST`	Maximum number of instances. Specifies the maximum number of instances of the module (10)
`RTC_MODULE_LANG`	Module Language Name. Describes which language has been used to develop the module (C++)
`RTC_MODULE_LANG_TYPE`	Module Language Type. Describes which type of language has been used to develop the module (COMPILE)
`RTC_MODULE_SPEC_END`	List End Marker. Indicates the end of the list. This item must be defined at the end of the list.

`RTC_MODULE_NAME`:Module Name

Specifies the name of the module (alphabet characters only). It is recommended to use a name as descriptive and specific as possible rather than very general words such as `Motor` or `Sensor`.

By default, the module name will automatically be used to set the component name. For example, the names of the components instantiated from a module called `HelloRTWorld` will be `HelloRTWorld0, HelloRTWorld1, ...` . Within the same manager, the component name is determined by appending the instance number (starting from 0) to the module name. Following this scheme, each component instance will be attributed a unique name

that, after being registered into the naming service, will be use to indentify the components. The component developer has the possibility to change the component naming scheme. This will be explain later on in this document.

RTC_MODULE_DESC:Component Description

Holds a short description of the module. We recommend the use of a concise and easy to understand description.

RTC_MODULE_VERSION:Module Version

Specifies the version numver of the module. A usual way of setting the version number is the "Major Version"."Minor Version" scheme. In this scheme, the "Major Version" number is incremented when the functionalities of the new version have been altered in a way that backward compatibility can not be preserved. For other benign revision, the "Minor Version" should be incremented. In addition to this two number scheme, a third "Revision Version" number can be used and incremented in case of simple bug fix.

RTC_MODULE_AUTHOR:Module Author

Specifies the name of the person and/or organization who developed the module.

RTC_MODULE_CATEGORY:Module Category

Specifies the category to which belongs to module. Presently, no particular rule concerning categories of modules has been defined. Such rule may be established in the future. For the time being, this field can be use freely.

RTC_MODULE_COMP_TYPE:Component Type

The 3 following component instantiation patterns are supported.

● **Component Type**

STATIC	Static Component. The component is instantiated only once and can not be destroyed nor reinstantiated. The number of instances of component directly controlling hardware is bound to the number of physical devices in the system. The STATIC type applies to component for which dynamic instantiation makes no sense.
UNIQUE	This type of component supports dynamic instantiation and destruction. However, as each instance of component has its own internal state, they cannot be interchanged.
COMMUTATIVE	This type of component supports dynamic instantiation and destruction. Moreover, as the instances of such component do not have specific states, they can be transparently interchanged

The value of this property can must be one of `STATIC, UNIQUE, COMMUTATIVE`, according to the type of component that is developed, enclosed in double quotation. The type of the `HelloRTWorld` component is `"COMMUTATIVE"`. At the present time, this setting does not influence the behavior of the component.

`RTC_MODULE_ACT_TYPE`:Activity Type

The 3 types of activity reflecting the behavioral pattern of the component are described below. (The OpenRTM-aist-0.2.0 implementation does not make use of this information. Presently, the user must develop of a real-time periodic process all by himself. In the future, we are planning to support the development of real-time periodic processes on top of real-time OS.

● **Activity Type**

PERIODIC	Periodic activity. The main activity is realized within a fixed-time periodic thread. This periodic thread can become a real-time periodic thread on top of a real-time OS.
SPORADIC	Sporadic activity. The main activity is also realized by a periodic thread but its cycle time is not fixed. For example, a component that operates only when the value of the sensor it controls changes is a sporadic component.
EVENT_DRIVEN	Event driven activity. The component only reacts to requests from external entities, like calls to its CORBA interface from other components.

The type of activity of the `HelloRTWorld` component is `"SPORADIC"`.

RTC_MODULE_MAX_INST:**Maximum number of instances.**

Specifies the maximum number of instances of the module. If the component is of the STATIC type, the maximum number of instances are constructed when the module is initialized.

RTC_MODULE_LANG: **Module Language Name and Type**

Describes which programming language and programming language type have been used to develop the module.

● **Module Language Name and Type**

Module Language Name	‘‘C++’’, ‘‘Python’’, ‘‘Ruby’’, ‘‘Tcl’’
Module Language Type	‘‘COMPILE’’, ‘‘SCRIPT’’

Presently, both the C++ and Python languages are supported for the development of RT-Components. The Language Type should be set to ”COMPILE” if C++ is used, ”SCRIPT” in the case of Python.

The specification of the component profile may change in the future. We are presently considering the possibility of importing the profile of a component from an XML file.

3.3.2 The Source File

Next, we will write the code for the main process, the factory and the initialization functions.

HelloRTWorld.cpp

```
#include "HelloRTWorld.h"
#include <iostream>

using namespace std;

HelloRTWorld::HelloRTWorld(RtcManager* manager)
  : RtcBase(manager)

{
}

RtmRes HelloRTWorld::rtc_active_do()
{
  std::cout << "Hello RT World!!!" << std::endl;
  return RTM_OK;
}

extern "C" {

  RtcBase* HelloRTWorldNew(RtcManager* manager)
  {
    return new HelloRTWorld(manager);
  }


  void HelloRTWorldDelete(RtcBase* p)
  {
```

```
    delete (HelloRTWorld *)p;
    return;
  }

  void HelloRTWorldInit(RtcManager* manager)
  {
    RtcModuleProfile profile(hellortworld_spec);
    manager->registerComponent(profile, HelloRTWorldNew, HelloRTWorldDelete);
  }
};
```

The pointer to the instance of the component manager passed to the constructor is forwarded to the base class `RtcBase` and the RtcBase is initialized. Do not forget the initialization of `RtcBase`.

Next, we find the only method called `RtmRes HelloRTWorld::rtc_active_do()`. This is the main processing part of an RT-Component.

The main processing part of the component is called the **Activity** of the component. The activity of the component can hold different states that change following well defined state transitions. After the start up phase of the component, the component activity will finally reach the state `ACTIVE` and execute the `rtc_active_do()` method in loop.

Below is the list of states supported by the component activity.

● **Component Activity States**

`UNKNOWN` :	Reflects that the state of the activity is unknown. A component itself will never go through this state, but it can come in handy when a component must process the state of another component to which the connection has failed or for monitoring.
`BORN` :	State of a component that has just been instantiated.
`INITIALIZING` :	Indicates that the component is initializing.
`READY` :	Indicates that the component is ready to process request from other components.
`STARTING` :	Transitional state between the READY and ACTIVE states.
`ACTIVE` :	Indicates that the component is active and that its main processing loop is executing.
`STOPPING` :	Transitional state between the ACTIVE and READY states.
`ABORTING` :	Transitional state between the ACTIVE and ERROR states after an error during the execution of the main processing loop occurred.
`ERROR` :	Indicates that a recoverable error has occurred.
`FATAL_ERROR` :	Indicates that an unrecoverable error has occured.
`EXITING` :	Indicate that the component is terminating. Transitional state while the component releases its resources.

The activity state transition follows the logic described in figure 3.7.

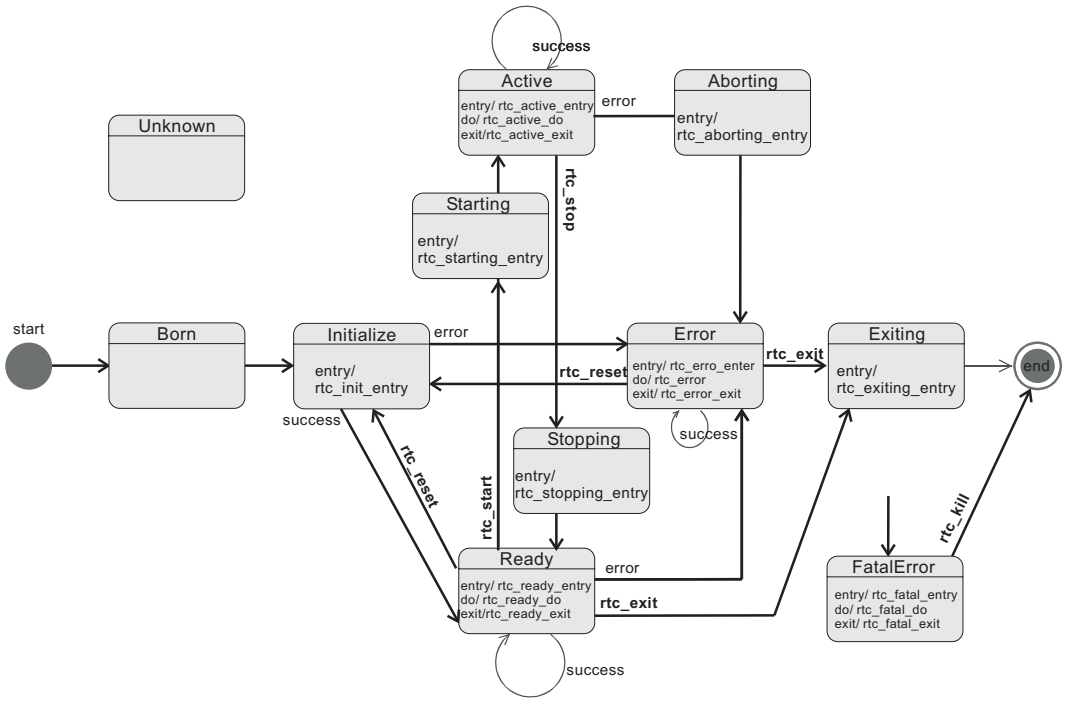

Fig. 3.7: State transitions of an RT-Component

To each of the above defined states correspond a set of supporting method. For example, in the case of the `RTC_READY` state, the associated functions are named `rtc_ready_???()`. The `???` part reflects the timing of the state transition.

`entry` is executed only once when the activity enters the given state from another state `exit` is executed only once when the activity leaves the given state to another state. `do` is executed periodically while in the given state.

● Detail of the State Transitions

entry	:	is executed only once when the activity enters the given state from another state
do	:	is executed periodically while in the given state.
exit	:	is executed periodically while in the given state.

The states can roughly be classified int 2 categories : the steady states and the transitional states.

● Steady and Transitional States

Steady States	`READY, ACTIVE, ERROR, FATAL_ERROR`
TransitionalStates	`BORN, INITIALIZING, STARTING STOPPING, ABORTING, EXITING`
Other	`UNKNOWN`

Once entering into a steady state, the activity will remain in that state until some action happens (like activation, deactivation, reinitialization, termination...). During this time, the

rtc_xxx_do() method will be executed in loop. When entering and exiting a steady state, the rtc_xxx_entry() and rtc_xxx_exit() method are executed respectively.

On the other hand, upon entering a transitional state, the rtc_xxx_entry() method will be executed but the activity will then immediately enter into the next state. For this reason, no rtc_xxx_do() and rtc_xxx_exit() method are associated with transitional states.

All the methods associated with the states described above are defined as virtual methods within RtcBase, the base class for all RT-Components. Therefore, a component developer only needs to override the associated methods of states which transition need special processing.

The state methods defined in the base class RtcBase are listed below.

RtcBase
+ virtual RtmRes rtc_init_entry()
+ virtual RtmRes rtc_ready_entry()
+ virtual RtmRes rtc_ready_do()
+ virtual RtmRes rtc_ready_exit()
+ virtual RtmRes rtc_starting_entry()
+ virtual RtmRes rtc_active_entry()
+ virtual RtmRes rtc_active_do()
+ virtual RtmRes rtc_active_exit()
+ virtual RtmRes rtc_stopping_entry()
+ virtual RtmRes rtc_aborting_entry()
+ virtual RtmRes rtc_error_entry()
+ virtual RtmRes rtc_error_do()
+ virtual RtmRes rtc_error_exit()
+ virtual RtmRes rtc_fatal_entry()
+ virtual RtmRes rtc_fatal_do()
+ virtual RtmRes rtc_fatal_exit()
+ virtual RtmRes rtc_exiting_entry()

In the case of our sample component, as we simply want "HelloRTWorld!!!" be continuously displayed while in the active state, we define rtc_active_do() as follow :

HelloRTWorld::rtc_active_do()

```
RtmRes HelloRTWorld::rtc_active_do()
{
  std::cout << "Hello RT World!!!" << std::endl;
  return RTM_OK;
}
```

Here, the return value RtmRes (RTM result) is RTM_OK .

A return value of the type RtmRes can take 4 possible values : RTM_OK, RTM_ERR, RTM_WARNING, RTM_FATAL_ERR . The meaning of each value is explained below.

RtmRes

RTM_OK	Normal termination. The current state is maintained
RTM_ERR	An error occurred. The activity will switch to the Error state. If the current state is Active, the activity will switch to the Error state via the transitional Aborting state.
RTM_WARNING	A warning was issued. Indicates that the method did not execute as it was expected to but without serious consequence for the whole activity. The current state can therefore be maintain - but some attention be be required
RTM_FATAL_ERR	A fatal error occurred. The activity will switch to the Error state. If the current state is Active, the activity will switch to the Error state via the transitional Aborting state.

As you can see, the RtmRes value returned by the method will influence the state of the activity by initiating a state transition. In our example, the method always return RTM_OK.

3.3.3 Component Executable Files

In order to build a standalone version executable file of the above described component, we must define the main function in the source code. In our case, we first create the component manager in which we then instantiate the component and activate it. The source code resides in the file HelloRTWorldComp.cpp described below.

The main function will sequentially instantiate, intialize and activate the component manager, instantiate the component and finally run the manager.

The manager class constructor requires the command line arguments as parameter. We will therefore directly call it using the command line arguments received by the main function.

initManager() will initialize the manager, then activateManager() will activate it and register it to the naming service.

By calling initModuleProc(), we provide the manager with the pointer to a function which itself requires a pointer to the manager as argument. The manager can then call this function at the appropriate time. Finally, we start the manager by calling runManager().

The MyModuleInit() function will initialize the component by calling the HelloRTWorldInit() initialization function. Then, the createComponent() function will create 1 instance of the component within the manager. To do so the createComponent() function requires the module name of the component as its first agument (in our example, it is "HelloRTWorld") and the component category type as its second argument.

Finally, by calling its rtc_start() method, we activate the component which state will switch to ACTIVE. The call to usleep(5000) will make the manager wait until the component that has been instantiated reaches the READY state. In our example, the creation and activation of the component occurs within the MyModuleInit() function. However, in general, the

creation and activation are taken care of by external means (such as the use of rtc-link).

HelloRTWorldComp.cpp

```
#include <rtm/RtcManager.h>
#include <string>
#include "HelloRTWorld.h"


void MyModuleInit(RtcManager* manager)
{
  HelloRTWorldInit(manager);

  std::string name;
  RtcBase* comp;
  comp = manager->createComponent("HelloRTWorld", "Generic", name);
  usleep(5000);
  comp->rtc_start();

}


int main (int argc, char** argv)
{
  RTM::RtcManager manager(argc, argv);
  // Initialize manager
  manager.initManager();
  // Activate manager and register to naming service
  manager.activateManager();
  // Initialize my module on this maneger
  manager.initModuleProc(MyModuleInit);
  // Main loop
  manager.runManager();
  return 0;
}
```

3.3.4 Writing the makefile and building the project

From the 3 files `HelloWorld.cpp, HelloWorld.h, HelloWorldComp.cpp` described above, we will build a standalone component as well as a loadable module. The makefile used to build the component should look like the code below.

The value of `CXX_FLAGS` and `LD_FLAGS` may vary depending on the environment in which OpenRTM is build. In this example, the compile and linking options are adapted to build OpenRTM-aist on top of omniORB. However the value of these options may differ according to where the dependency packages ACE, omniORB and boost have been installed. It is therefore necessary to set these options in accordance to one's environment. In our next example, we will describe a more generic approach to building components but for the time being just consider the setting of the compile and linking options necessary.

Makefile

```
CXXFLAGS = -I/usr/local/include -I/usr/local/include/rtm/idl
```

```
LDFLAGS  = -L/usr/local/lib -lpthread -lACE -lboost_regex -lomniORB4 \
           -lomnithread -lomniDynamic4 -lRTC
SHFLAGS  = -shared
.SUFFIXES: .cpp .o .so

all: HelloRTWorldComp HelloRTWorld.so

.cpp.o:
rm -f $@
$(CXX) $(CXXFLAGS) -c -o $@ $<

.o.so:
rm -f $@
$(CXX) $(SHFLAGS) -o $@ $< $(LDFLAGS)

HelloRTWorldComp: HelloRTWorld.o HelloRTWorldComp.o
$(CXX) -o $@ HelloRTWorld.o HelloRTWorldComp.o $(LDFLAGS)

clean:
rm -f *~ *.o *.so *Comp
```

Moreover, as the `.o.so` rule was set in this Makefile, the Loadable Module Component will also be generated. This component can be used after loading it in the component server `rtcd` provided by OpenRTM-aist. For more details, refer to the chapter explaining to the use of the tools in OpenRTM-aist.

3.3.5 Execution - Testing

Using the Makefile described above to build the source code, we could generate the executable file called `HelloRTWorldComp`. In order to execute the file, we must provide the name of the initialization file containing the component settings using a command-line option `-f rtc.conf`. Upon execution the output of the component should look like Figure 3.8.

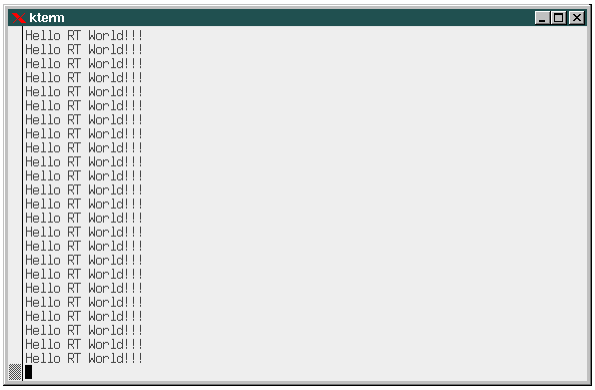

Fig. 3.8: Execution of HelloRTWorldComp

3.4 Writting a component using rtc-template

As you could see so far, the code necessary to create a component is already well defined. OpenRTM-aist comes with an automatic code generator called `rtc-temlpate`. From now, we will explain how to use `rtc-template` to develop components.

By feeding the tool with the name and type of the component and its InPort/OutPort, the C++ (or Python) skeleton code can be automatically generated. The component developer can then customize the generated code to suit his particular needs by overriding the necessary activity method.

3.4.1 Specifying the Component

First, we must define our component. In order to generate the component, we need to specify the following items.

- Module Name of the component
- Category of the component
- Type of the component
- Activity Type of the component
- Number of InPorts, their name and type
- Number of OutPorts, their name and type

The "Module Name of the component", "Category of the component", "Type of the component", "Activity Type of the component" will be used to specify the `RTC_MODULE_NAME`, `RTC_MODULE_CATEGORY`, `RTC_MODULE_COMP_TYPE`, `RTC_MODULE_ACT_TYPE` fields of the `RtcModuleProfile` structure found in the header file of the component (See the HelloRTWorld sample).

Module Name of the Component

The component name will be used by other applications and component to search for the component. To avoid that the name of the component collides with the name of another component, it is preferable to use a very concrete and specific name. As of today, there is no special guideline to set the name of the component. However, if some users were to establish such guideline, we may include it in a future version.

● **Example of Module Name of a Component**

PA10	Manipulator component
NittaFTSensor	Torque Sensor component
ForceControllerForPA10	Force Controller for PA10 component
:	:

Category Name of the component

Specifies to which category belongs the module. The category should be set accordingly to the functionality offered by the component. As of today, there is no special guideline to set the category of the component. However, if some users were to establish such guideline, we may include it in a future version.

● **Example of Category Name of a component**

Manipulator	for a manipulator
FTSensor	for a torque sensor
Controller	for a controller
:	:

Component Type

Specifies the type of instantiation pattern of the component. The following options are available.

● **Component Types**

STATIC	The component is instantiated once when registered to the manager. The instantiation of the component is restricted to one. This type is particularly suited to component directly dealing with hardware.
UNIQUE	Dynamic instantiation and destruction of the component is possible. However, As component0 and component1 may have specific internal states, they cannot be used interchangeably.
COMMUTATIVE	These components can be used interchangeably. Thi type of component is especially suited for raw software logic.

Type of Component Activity

The following types of component internal activity are supported.

● **Types of Component Activity**

PERIODIC	The main activity is realized within a fixed-time periodic thread. This periodic thread can become a real-time periodic thread on top of a real-time OS such as ART-LINUX.
SPORADIC	The main activity is also realized by a periodic thread but its cycle time is not fixed.
EVENT_DRIVEN	The component only reacts to requests from external entities, like calls to its CORBA interface from other components.

Number, Name and Variable Type of an InPort

An InPort is the mean by which a component receives external data. A component can manage several InPorts. An InPort can be referenced by external applications or components by the name it was attributed.

Name

● **Examples of name for an InPort**

"velocity"	Velocity
"reference"	Reference
"position"	Position

Variable Type

InPorts can support several pre-defined variable types. A "single" and "sequence" (array of the same variable type) version of each variable type is supported.

● **InPort/OutPort Variable Types**

TimedShort	Timestamped Signed Short Int variable
TimedLong	Timestamped Signed Long Int variable
TimedUShort	Timestamped Unsigned Short Int variable
TimedULong	Timestamped Unsigned Long Int variable
TimedFloat	Timestamped Float variable
TimedDouble	Timestamped Double variable
TimedChar	Timestamped Char variable
TimedBoolean	Timestamped Boolean variable
TimedOctet	Timestamped Octet variable
TimedString	Timestamped String variable

● **InPort/OutPort Sequenced Variable Types**

TimedShortSeq	Timestamped sequence of Signed Short Int
TimedLongSeq	Timestamped sequence of Signed Long Int
TimedUShortSeq	Timestamped sequence of Unsigned Short Int
TimedULongSeq	Timestamped sequence of Unsigned Long Int
TimedFloatSeq	Timestamped sequence of Float
TimedDoubleSeq	Timestamped sequence of Double
TimedCharSeq	Timestamped sequence of Char
TimedBooleanSeq	Timestamped sequence of Boolean
TimedOctetSeq	Timestamped sequence of Octet
TimedStringSeq	Timestamped sequence of String

In OpenRTM-aist, these types are defined in a CORBA compatible way in the file `RTCDataType.idl`. If necessary, the user can define custom structures and append them to `RTCDataType.idl`. At this point, we will not explain the details of this operation.

Number, Name and Variable Type of an OutPort

An OutPort is the mean by which a component makes available its data to external entities. A component can manage several InPorts. An InPort can be referenced by external applications or components by the name it was attributed.

Name

● **Examples of name for an OutPort**

"velocity"	Velocity
"reference"	Reference
"position"	Position

Variable Type

OutPorts can support the same pre-defined variable types as InPorts can. InPorts and OutPorts can exchange data only if they support the same variable type (Figure 3.9).

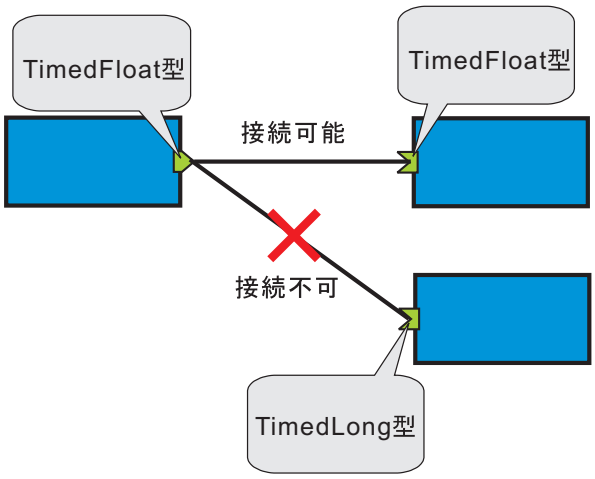

Fig. 3.9: Relation between the InPort and OutPort types

3.4.2 Generation of the skeleton of an RT-Component using rtc-template

Once the above mentioned items are specified, the skeleton of the component can be generated using the component template generator `rtc-template`.

Here, we will generate a component with the following characteristics.

Activity Process

We will assume that our component will use a library implementing the basic routines necessary for the control of a manipulator and offering the following functions.

● Example of functions offered by the manipulator control library

manipulator_init()	: Initialization of the manipulator
manipulator_on()	: Turn ON the manipulator servo
manipualtor_off()	: Turn OFF the manipulator servo
manipulator_setpos()	: Set the target point of the manipulator's end-effector
manipulator_getpos()	: Retrieves the posture of the manipulator
manipulator_emgstop()	: Emergency stop of the manipulator
manipulator_destroy()	: Release the resources of the manipulator

● Specifications of the Component

Component Module Name	MyManipulator
Component Category	Manipulator
Component Type	STATIC
Component Activity Type	PERIODIC
Number of InPorts	1 : Receives the target position
InPort Name	"pos"
InPort Variable Type	TimedFloatSeq
Number of OutPorts	1 : Ouputs the present position
OutPort Name	"pos"
OutPort Variable Type	TimedFloatSeq

Generation of the skeleton using rtc-template

We first have to make the directory in which the component will be generated. There is no restriction on the path of the directory.

```
> cd
> mkdir MyManipulator
> cd MyManipulator
```

Let's first take a look at the "help" section.

```
> rtc-template --help

Usage: rtc-template [OPTIONS]
Options:
    [--help]                              Print this help.
    [--c++]                               Generate C++ template code.
    [--python]                            Generate Python tempalte code.
    [--output[=output_file]]              Output base file name.
    [--module-name[=name]]                Your module name.
    [--module-desc[=description]]         Module description.
    [--module-version[=version]]          Module version.
    [--module-author[=author]]            Module author.
    [--module-category[=category]]        Module category.
    [--module-comp-type[=component_type]] Component type.
    [--module-act-type[=activity_type]]   Component's activity type.
    [--module-max-inst[=max_instance]]    Number of maximum instance.
    [--module-lang[=language]]            Language.
    [--module-inport[=PortName:Type]]     InPort's name and tyoe.
    [--module-outport[=PortName:Type]]    OutPort's name and type
       :
Example:
    rtc-template --c++ --module-name=Sample --module-desc='Sample component' \
    --module-version=0.1 --module-author=DrSample --module-category=Generic \
    --module-comp-type=COMMUTATIVE --module-act-type=SPORADIC \
    --module-max-inst=10  \
    --module-inport=Ref:TimedFloat --module-inport=Sens:TimedFloat \
    --module-outport=Ctrl:TimedDouble --module-outport=Monitor:TimedShort
```

By passing the settings of the component we want to build as arguments to `rtc-template`, the skeleton of the component will automatically be generated. We will now generate the component we have just specified.

```
> rtc-template --c++ --module-name=MyManipulator \
    --module-desc='My simple manipulator' \
    --module-version=0.1 --module-author=MyName --module-category=Manipulator \
    --module-comp-type=STATIC --module-act-type=PERIODIC \
    --module-max-inst=1  \
    --module-inport=posin:TimedFloatSeq --module-outport=posout:TimedFloatSeq
MyManipulator.h was generated.
MyManipulator.cpp was generated.
MyManipulatorComp.cpp was generated.
Makefile.MyManipulator was generated.
> ls
Makefile.MyManipulator  MyManipulator.h        MyManipulatorComp.cpp
MyManipulator.cpp       MyManipulator.h
MyManipulator.cpp       MyManipulatorComp.cpp
```

The operation described above will generate the C++ code of the component, as well as its associated Makefile.

```
> make -f Makefile.MyManipulator
```

or also

```
> mv Makefile.MyManipulator Makefile
```

Let's now call the make command.

```
> make
rm -f MyManipulator.o
g++ `rtm-config --cflags` -c -o MyManipulator.o MyManipulator.cpp
     :
     :
g++ `rtm-config --libs` -o MyManipulatorComp MyManipulator.o MyManipulatorComp.o
rm -f MyManipulator.so
g++ -shared `rtm-config --libs` -o MyManipulator.so MyManipulator.o
> ls
Makefile.MyManipulator  MyManipulator.o     MyManipulatorComp.cpp
MyManipulator.cpp       MyManipulator.so*   MyManipulatorComp.o
MyManipulator.h         MyManipulatorComp*
```

Here, the loadable module version(`MyManipulator.so`) as well as the executable version(`MyManipulatorComp`) of the component have been generated. However, as the activity has not been defined, the component will not perform any work yet. We now have to complete the development of the component by inserting the process logic code into the skeleton.

3.4.3 Let's take a look at the component source code

Take a look now at the code of the component generated by `rtc-template`. The source code looks very much like the one of the HelloRTWorld, except for the component name and the InPort/OutPort parts that have been adapted to our settings.

● **Files generated by `rtc-template`**

`MyManipulator.h`	MyManipulator component header file.
`MyManipulator.cpp`	MyManipulator component source file
`MyManipulatorComp.cpp`	Source file to run the MyManipulator component in standalone mode.
`Makefile`	Makefile to build the component.

For example, the `MyManipulator.h` header file should look like below.

MyManipulator.h

```
class MyManipulator
  : public RTM::RtcBase
{
 public:
  MyManipulator(RtcManager* manager);

  //  virtual RtmRes rtc_ready_entry();
  //  virtual RtmRes rtc_ready_do();
  //  virtual RtmRes rtc_ready_exit();
  //  virtual RtmRes rtc_active_entry();
  virtual RtmRes rtc_active_do();
  //  virtual RtmRes rtc_active_exit();
  //  virtual RtmRes rtc_error_entry();
  //  virtual RtmRes rtc_error_do();
  //  virtual RtmRes rtc_error_exit();
  //  virtual RtmRes rtc_fatal_entry();
  //  virtual RtmRes rtc_fatal_do();
  //  virtual RtmRes rtc_fatal_exit();
  //  virtual RtmRes rtc_init_entry();
  //  virtual RtmRes rtc_starting_entry();
  //  virtual RtmRes rtc_stopping_entry();
  //  virtual RtmRes rtc_aborting_entry();
  //  virtual RtmRes rtc_exiting_entry();

  TimedFloatSeq m_posin;
  InPortAny<TimedFloatSeq> m_posIn;
  TimedFloatSeq m_posout;
  OutPortAny<TimedFloatSeq> m_posOut;

};
```

The rather long serie of `rtc_xxx_yyy` methods are the method the support the state transitions of the component. Refer to figure 3.7 (p.36) describing the state machine or to its related explanation for more details about state transitions.

3.4.4 Deciding which activity to use

Keeping in mind the state transitions described in figure 3.7, we will decide what processing will take place in each activity. While considering the functionalities and states supported

by the aforementionned library, you should now allocate, to each state, the processing that you want to do. Remember that the initialization process should take place while in state `INITIALIZING`, that the process defined for the state `STARTING` will be executed just before the component enters the `ACTIVE` state, that main processing loop will execute while in the `ACTIVE` state, and that the process defined for the state `STOPPING` will be executed when the component will switch from state `ACTIVE` back to state `READY`.

To each state of our MyManipulator component, we will assign the processing to be executed in the following way.

● **Details of the processing**

`rtc_init_entry()`	We will call the `manipulator_init()` functin to initialize the library. We will assume that the boolean return value of `manipulator_init()` indicates if the method succeeded or not.
`rtc_starting_entry()`	We will call the `manipulator_on()` function to turn the servo of the manipulator ON. We will assume that the boolean return value of `manipulator_on()` indicates if the method succeeded or not.
`rtc_stopping_entry()`	We will call the `manipulator_off()` function to turn the servo of the manipulator OFF. We will assume that the boolean return value of `manipulator_off()` indicates if the method succeeded or not.
`rtc_active_do()`	We will first call `manipulator_setpos()` to set the target point of the manipulator, using the data received by the InPort, and then, get the actual position of the end-effector by calling `manipulator_getpos()` and output it to the OutPort.
`rtc_aborting_entry()`	As an error occurred while in `ACTIVE` state, we will call the `manipulator_emgstop()` function for the manipulator to make an emergency stop.
`rtc_exiting_entry()`	Termination of the component. We will call the `manipulator_destroy()` function to release the resources

3.4.5 Implementation

Once we have decided what should happen in each state as described above, we have to actually implement each state method.

In order to develop a safe component, it is very important to specify what action should be taken in the case an error occurred. We have to seriously consider which cases should lead to the `ERROR` state and which cases should lead to the `FATAL_ERR` state.

Implementation of the method : `rtc_init_entry()` of the state `RTC_INITIALIZING`

MyManipulator.h

```
virtual RtmRes rtc_init_entry();
```

We just uncommented out the method definition in the header skeleton.

MyManipulator.cpp

```
RtmRes MyComponent::rtc_init_entry()
{
  if (!manipulator_init())
  {
    return RTM_ERR;
  }
  return RTM_OK;
}
```

Here, we just check the value returned by the call to `manipulator_init()`. If the return value is true, we return `RTM_OK`, otherwise, we return `RTM_ERR`. The possible values a state method can return are listed below.

● **State method completion and return value**

`RTM_OK` :	Normal Completion
`RTM_ERR` :	An Error occured before Completion
`RTM_FATAL_ERR` :	A Fatal Error occured before Completion

In case of a normal completion, the component makes a transition to its next state. In this case, the component will make a transition to the state `RTC_READY`. If an error occurred before completion, the component goes to the `ERROR` state. When in error state, the component can be re-initialized from outside and will then go back to the `RTC_INITIALIZE` state. In the case of a fatal error, recovery is not possible. It is therefore possible to force the termination of a component by simulating a Fatal Error.

Implementation of the method `rtc_starting_entry()` of the state `RTC_STARTING`

MyManipulator.h

```
virtual RtmRes rtc_starting_entry();
```

We just uncommented out the method definition in the header skeleton.

MyManipulator.cpp

```
RtmRes MyComponent::rtc_starting_entry()
{
  if (!manipulator_on())
  {
    return RTM_ERR;
  }
  return RTM_OK;
}
```

Depending on the value returned by manipulator_on(), we return either RTM_OK or RTM_ERR.

Implementation of the method rtc_stopping_entry() of the state RTC_STOPPING

MyManipulator.h

```
virtual RtmRes rtc_stopping_entry();
```

We just uncommented out the method definition in the header skeleton.

MyManipulator.cpp

```
RtmRes MyComponent::rtc_stopping_entry()
{
  if (!manipulator_off())
  {
    return RTM_ERR;
  }
  return RTM_OK;
}
```

Depending on the value returned by manipulator_on(), we return either RTM_OK or RTM_ERR.

Implementation of the method rtc_active_do() of the state RTC_ACTIVE

MyManipulator.h

```
virtual RtmRes rtc_active_do();
```

We just uncommented out the method definition in the header skeleton.

Next, we will implement rtc_active_do().

In rtc_active_do(), we will first call manipulator_setpos() to set the target point of the manipulator, using the data received by the InPort, and then, get the actual position of the end-effector by calling manipulator_getpos() and output it to the OutPort. We can get data from the InPort by calling m_posIn.read().

MyManipulator.cpp

```
RtmRes MyComponent::rtc_active_do()
{
  float pos_in[6];
  float pos_out[6];

  // Read data from the InPort
  // InPort からデータ読み込み
  m_posin = m_posIn.read();

  // Verify that the data from the InPort is of the proper format
  // InPort に規定数のデータが入っているか確認
  if (m_posin.data.length() == 6)
  {
    for (int i = 0; i < 6; i++)
    {
       pos_in[i] = m_posin.data[i];
    }
    // Set the target position
    // 位置データをセット
    if (!manipulator_setpos(pos_in))
    {
      return RTM_ERR;
    }
  }

  // Acquire the present actual position
  // 現在位置を取得
  if (!manipulator_getpos(pos_out))
  {
    return RTM_ERR+
  }

  // Output it to the OutPort
  // OutPort に出力
  m_posout.data.length(6);
  for (int i = 0; i < 6; i++)
  {
     m_posout.data[i] = pos_out[i];
  }
  m_posOut.write(m_posout);

  return RTM_OK;
}
```

The members of the TimedFloatSeq Data Type (as well as other sequence types) structure are the time variable `tm` and the data variable `data`. As `data` is a sequence, the following method are used to access specific elements.

● Access to data elements from an InPort/OutPort

`m_poin.data[1]`	Access to the data specified by the index
`m_posin.data.length()`	Retrieves the number of elements in data
`m_posin.data.length(10)`	Sets the number of elements data can hold

In the same way for the OutPort, after acquiring the position of the end-effector on the manipulator by using `manipulator_getpos()`, and copying the data to `m_posout`, we output it to the OutPort using :

```
m_posOut.write(m_posout);
```

For more details on the methods supported by InPort and OutPort, refer to the manual or directly look into the source code.

Implementation of the method `rtc_aborting_entry()` of the state `RTC_ABORTING`

MyManipulator.h

```
    virtual RtmRes rtc_aborting_entry();
```

We just uncommented out the method definition in the header skeleton.

MyManipulator.cpp

```
RtmRes MyComponent::rtc_aborting_entry()
{
  if (!manipulator_emgstop())
  {
    return RTM_FATAL_ERR;
  }
  return RTM_OK;
}
```

In the case an error occurs while in the `RTC_ACTIVE` state, we will assume that the manipulator make an emergency stop. Here, in `rtc_aborting_entry()`, we will call `manipulator_emgstop()`. Depending on the value returned, we will return either `RTM_OK` or `RTM_ERR`.

Implementation of the method `rtc_exiting_entry()` of the state `RTC_EXITING`

MyManipulator.h

```
virtual RtmRes rtc_exiting_entry();
```

We just uncommented out the method definition in the header skeleton.

MyManipulator.cpp

```
RtmRes MyComponent::rtc_exiting_entry()
{
  manipulator_destroy();
  return RTM_OK;
}
```

We release the resources. As, from this state, it is only possible to terminate the component, we must return `RTM_OK`.

3.4.6 Building and executing the component

Now that we finished writting the source code, we rebuild the component. As we want to link the manipulator library, we may have to modify Makefile.MyManipulator.

```
> make -f Makefile.MyManipulator
```

Let's try to run the executable version of the component. A configuration file (usually called rtc.conf) is necessary to run the component. Refer to the `etcrtc.conf` file included in OpenRTM-aist for a complete description of the format of the configuration file. Here, we will create a simple configuration file in the current directory.

`NameServer Current Hostname:Port Number`

For example, Hostname : rtm.or.jp, Port Number : 6789. We then make the following input (exactly as written below).

```
> cat > rtc.conf
NameServer      rtm.or.jp:6789
\verb|^|D(Ctrl+D)
```

We can confirm by typing :

```
> cat rtc.conf
NameServer      rtm.or.jp:6789
```

Next, we start the CORBA Naming Service. The CORBA Naming Service can be started by inputting the following command :

`rtm-naming Port Number`

The port number passed as argument should be the same as the one defined in rtc.conf.

```
> rtm-naming 6789
Starting omniORB omniNames: ichi:9999
n-ando@ichi:/tmp/SampleComponent>
Fri Oct 29 17:12:51 2004:

Starting omniNames for the first time.
Wrote initial log file.
Read log file successfully.
Root context is IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f
4e616d696e67436f6e746578744578743a312e3000000100000000000000600000000010102000e00
00003135302e32392e39362e313638000f270b0000004e616d6553657276696365500020000000000
0000080000000100000000545441010000001c0000000100000001000100010000000100010509010
01000100000009010100
Checkpointing Phase 1: Prepare.
Checkpointing Phase 2: Commit.
Checkpointing completed.
```

Next, we will start the component. Unfortunately, as this component operates only when it receives target positions to its InPort, nothing will actually happen by just starting it. By following the procedure we just described, you can write the components that will issue the target positions and the component that will receive and process the current position. Finally, by connecting them you can compose the components into a working system.

```
> MyManipulatorComp -f rtc.conf
```

- 4- Tools in OpenRTM-aist

In this chapter, we will explain how to use the tools provided with OpenRTM-aist. OpenRTM-aist provides several development tools to support the development of RT-Components. At this point , it only includes basic tools. However, we are planning to extend to kit to respond to user's needs.

4.1 RTCLink

There are several ways to compose a system by combining RT-Components.

- Using a GUI
- Using XML
- Using Scripts
- By direct access from a component
- By direct access from an application

We will start by explaining the most intuitive and easiest way to compose a system using a graphical interface (RTCLink). Using this GUI, we can simply compose and manage a system by connecting InPorts and OutPorts in a drag and drop fashion and turn components ON and OFF by a simple click of the mouse. The ease of use of this method makes it very adapted to test single components or test a complete system.

4.1.1 Prerequisite

RTCLink was developed using wxPython, one of the tool kits for Python. Therefore, RTLink can be used on top of any OS, as long as it provides a Python runtime environment.

● **Prerequisites for running RTCLink**

Python	Python 2.3 or later
wxPython	wxPython 2.5.1.5u
OpenRTM	aist-0.2.0 or later

4.1.2 Starting RTCLink

After starting RTCLink, the window showed in figure 4.1 appears.

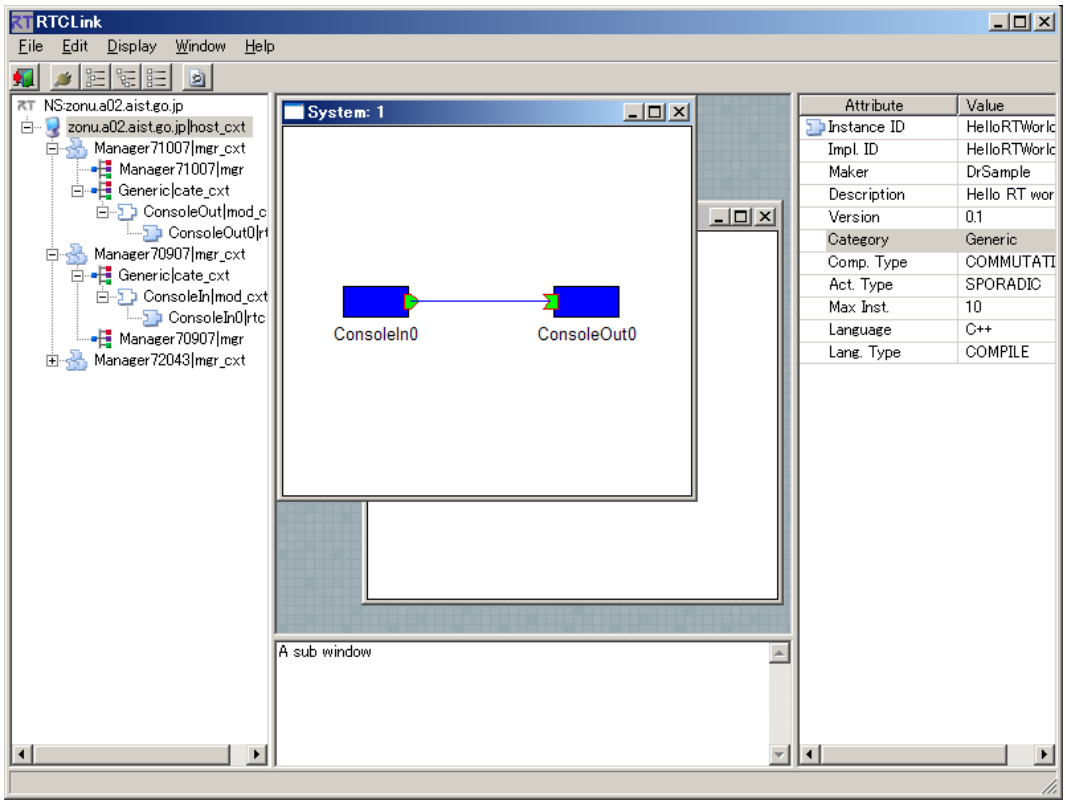

Fig. 4.1: RTCLink

The RTCLink user interface is basically composed of 4 main panes.

- The Naming Pane
- The Log Pane
- The Profile Pane
- The System Composer Pane

4.1.3 The Naming Pane

The left pane displays the list of components registered into the naming service in a tree like fashion. When a component starts, it will automatically register its name to a predefined naming service (The address of the naming service is usually set in the file named `rtc.conf`). You will see this name appear in the Naming Pane.

There are 2 ways of registering a name.

● Ways of registering a name

Long Name	Name generated from the hierarchy of contexts : Host Context / Manager Context / Category Context / Module Context / Component. Using this method, we can ensure the uniqueness of the name within the whole system. By default, this name is registered to the naming service.
Alias	Alias of the component. The user is free to set a name based on any hierarchical system. It is not even necessary to base the name on any hierarchy at all. However, the user must keep in mind the necessity of uniqueness of a name when deciding on a naming pattern. If a name conflict was to occur, the new object registration prevails and the old object becomes unretrievable.

For more detail on how to register aliases, refer to the `appendAlias()` methods in the `RtcBase` class reference.

4.1.3.1 Connection to the Name Server

At startup. RTCLink will connect to the default name server. The default name server is either the one to which RTCLink connected last (if the log has been saved), or the one running on the localhost itself.

The name server to which RTCLink is currently connected is displayed as the root element of the name tree.

To connect to a different name server, use the dialog displayed in figure 4.2 and which is accessible by either one of the following actions :

● Connecting to the name server

Click the  button in the toolbar
Select “File”]-”Connect Name Server” in the menu bar
Right click on the root item of the naming tree and select “Connect”in the contextual menu

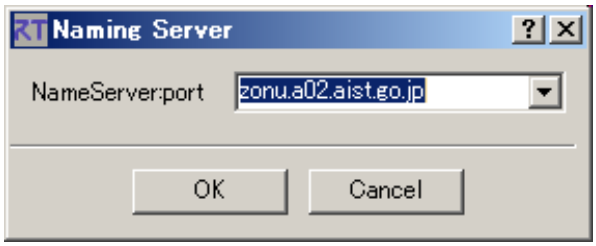

Fig. 4.2: Dialog for connecting to the name server

After inputting the address and the port number of the name server to which you want to connect using the `name_server_address:port_number` format, click on the OK button. If the port number is not specified, the default port number for the naming service defined in OmniORB will be used.

4.1.3.2 Item and Context menu

The following items are displayed in the naming tree.

 Host context
 Manager context
 Category context
 Module context
 Component

When right-clicking on any item in the naming tree, its associated context menu will appear.

For each type of item, the following context menus will appear.

● **Manager context (Not implemented)**

Create	Instantiate a component
Load	Load a module
Delete	Delete the corresponding naming context

● **Module context (Not implemented)**

Create	Instantiate a component
Delete	Delete the corresponding naming context

● **Component**

Start	Start the component
Stop	Stop the component
Reset	Reset the component
Exit	Terminates the component
Kill	Kill the component
Delete	Delete the corresponding naming context
Profile	Display the component profile

4.1.4 The Log Pane

In OpenRTM-aist, it is possible to register at the top level of the naming space in the naming service a reference to a component called “GlobalLogger”. The Global Logger can

collect the log entries issued by RT-Components. The log pane allows for real time monitoring of the log messages received by the Global Logger from all the components in the system.

4.1.5 The Profile Pane

The component profiles are displayed in this pane. The following entries of a profile can be displayed.

Instance ID	Component Instance ID (Component Name)
Implementation ID	Component Implementation ID (Module Name)
Maker	Component Developer's Name
Description	Brief description of the component
Version	Component Version
Category	Component Category
Component Type	Component Type
Activity Type	Activity Type
Max Instance	Maximum Number of Instances
Language	Language used to develop the component
Language Type	Type of the language used to develop the component
InPort Profile	InPort Profile
OutPort Profile	OutPort Profile

4.1.6 System Composer Pane

The central pane is used to compose the system. This pane is multi-windowed (using tabs in a UNIX environment).

● How to create a new System Composer Window

- Click the の  button in the toolbar
- Select "File"-"New System" in the menu bar

It is possible to open a new System Composer window by either clicking the  button in the toolbar, or selecting "File"-"New System" in the menu bar. It will then be possible to compose a new system by drag-and-dropping components from the Naming pane.

4.1.7 Compose a system in the System Composer Pane

4.1.7.1 Connecting Components

Let's expand the naming tree by clicking on it.

If the component you are looking for is not displayed in the naming tree, check the following points.

- Is the naming service running ? (Also check the port number)
- Is the component itself running ?

Drag a component in the tree by pressing the left button of the mouse and drop it in the System Composer Window. A new component block appears.

The block of a component that has not been started will be displayed in black.

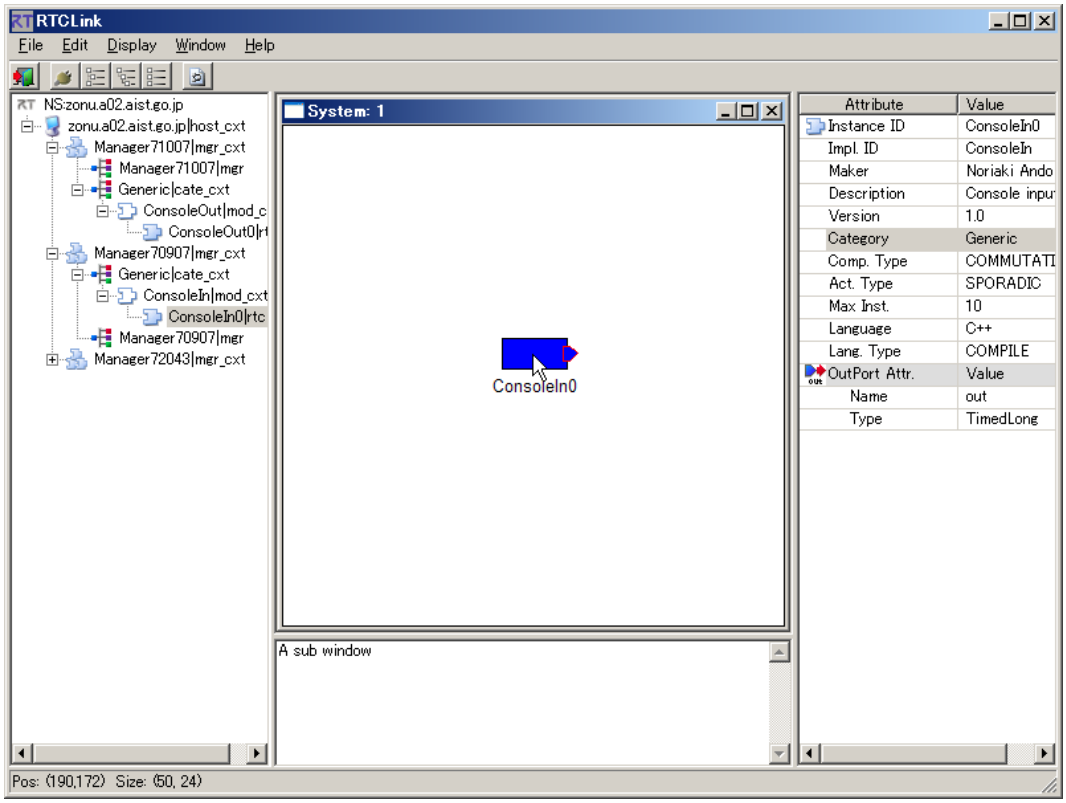

Fig. 4.3: Drag and drop for placing components. RTCLink remembers how components where places and connected from previous sessions.

4.1.7.2 Selecting, moving and deleting RT-Components

Selecting components A component can be selected with a left click on it(Figure 4.4). Several components can be select by pressing the "Shift Key" while selecting components. (Figure 4.5). The color of selected components changes.

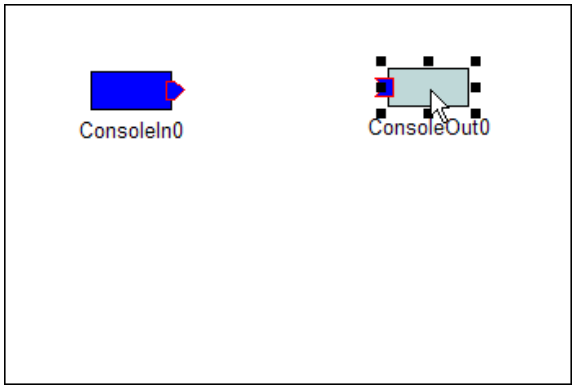

Fig. 4.4: Selecting a component

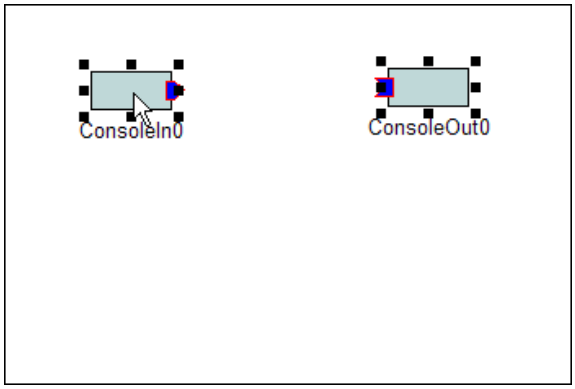

Fig. 4.5: Selecting several components

Moving components The selected component can be moved by dragging the mouse while pressing the left button (Figure 4.6). If several components are selected, start dragging the mouse from above one of the selected components. All the selected components will then move simultaneously.

Unselecting components An RT-Component can be unselected by a left-click in any other area (Figure 4.7).

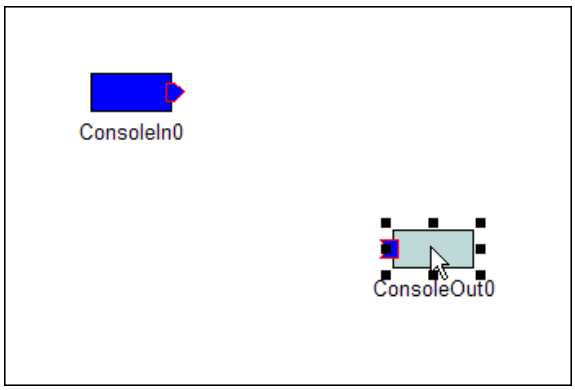

Fig. 4.6: Moving an RT-Component

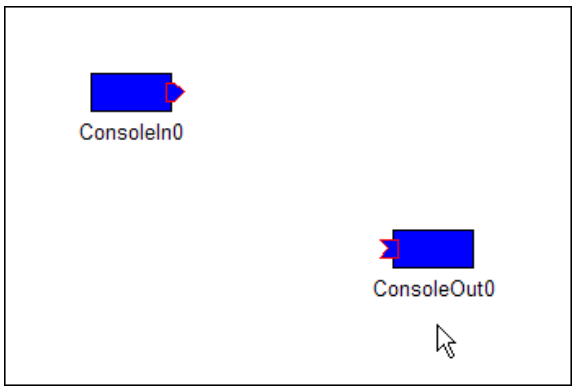

Fig. 4.7: Unselecting a component

Deleting components A right click above an RT-Component will open a context menu. By selecting 'Delete Item' in the menu or by pressing the "Delete Key", the RT-Component under the cursor will be deleted from the System Composer Window (Figure 4.8). To delete several components, they first must be selected. Then, a right click in an area different from the selected components will open the context menu. By selecting 'Delete to Selected Item' or by pressing the "Delete Key", all the selected components will be deleted from the System Composer window. This operation only deletes RT-Components from the System Composer Window. The components themselves are not actually deleted.

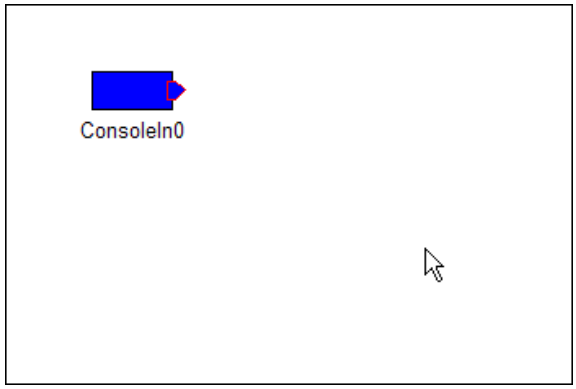

Fig. 4.8: Deleting a component

Flipping components horizontally A middle click on a component will flip the component horizontally, inverting the placement of its InPorts and OutPorts (Figure 4.9).

Rotating components A middle click on a component while pressing the "Shift Key" will rotate the component, reorienting its InPorts and OutPorts in the vertical direction. (Figure 4.10).

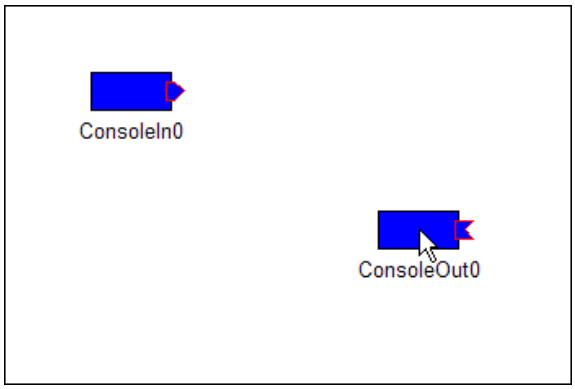

Fig. 4.9: Flipping a component horizontally

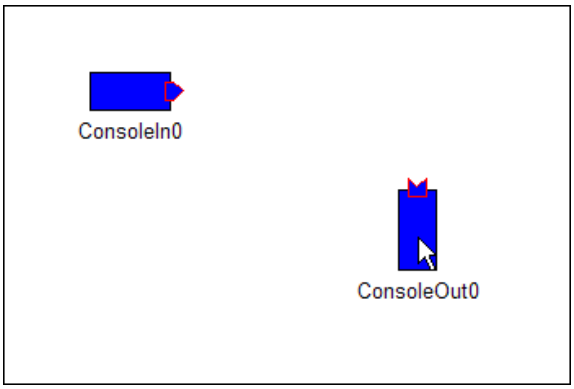

Fig. 4.10: Rotating a component

Resizing components A component can be resized by, after selecting it, dragging one of the black handles at each of its corners (Figure 4.11).

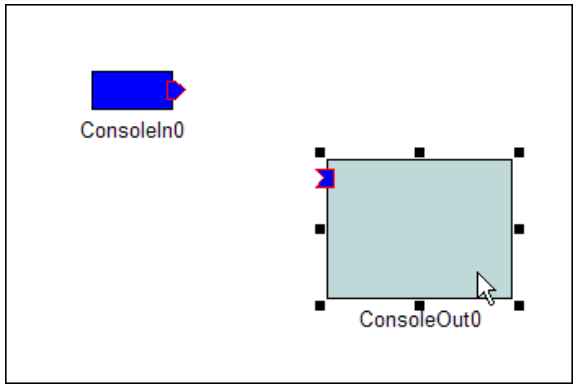

Fig. 4.11: Resizing a component

4.1.7.3 Displaying InPorts/OutPorts name and type

A right click on an InPort/OutPort will display its name and type (Figure 4.12).

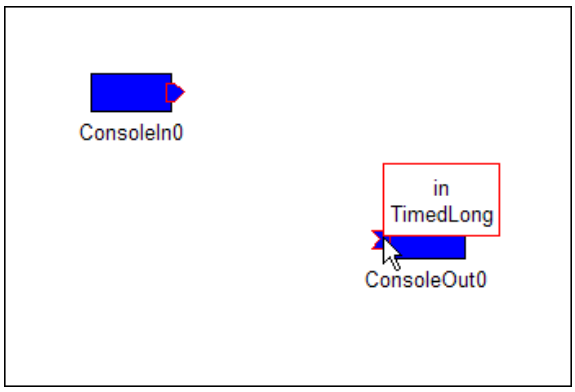

Fig. 4.12: Displaying the name and type of an InPort/OutPort

4.1.7.4 Connecting and disconnecting components

There are 2 ways to interconnect components described below.

Connecting an InPort/OutPort with a click To connect an InPort to an OutPort, fist make left click on the origin Inport (Figure 4.13), then a left click on the target OutPort (Figure 4.14). A line will be drawn between the 2 newly connected InPort and OutPort and their color will change.

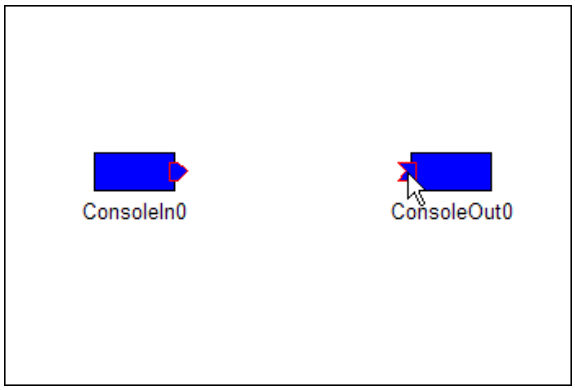

Fig. 4.13: Connecting components 1(a)

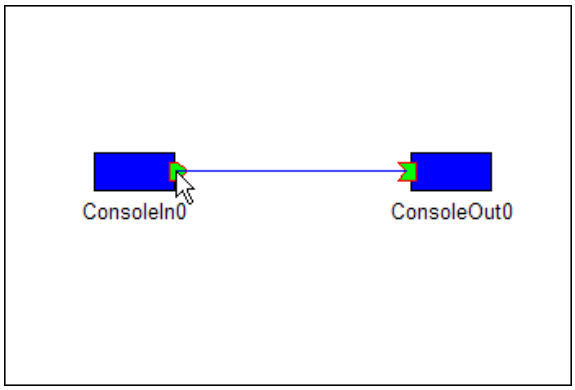

Fig. 4.14: Connecting components 1(b)

Connecting an InPort/OutPort with a drag and drop Another way to connect an InPort to an OutPort is to drag the origin InPort and drop it onto the target OutPort (Figure 4.15). A line will be drawn between the 2 newly connected InPort and OutPort and their color will change (Figure 4.16).

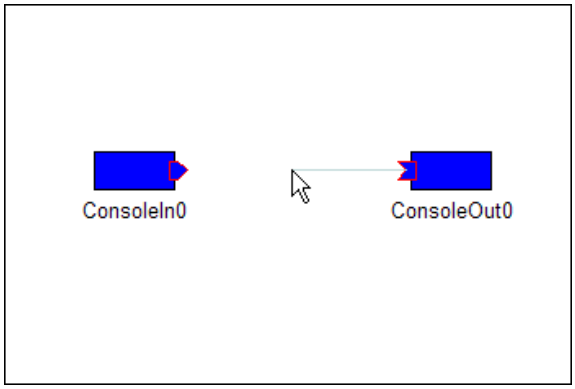

Fig. 4.15: Connecting components 2(a)

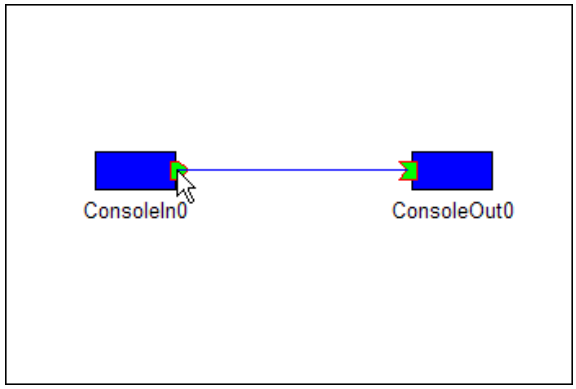

Fig. 4.16: Connecting components 2(b)

Disconnecting components After a left click on the line connecting the 2 components, press the "Delete Key" (Figure 4.18).

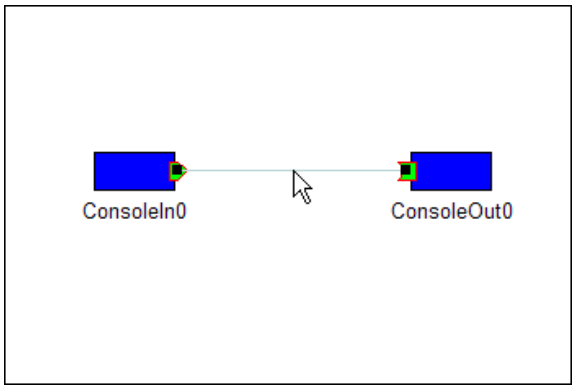

Fig. 4.17: Disconnecting components(a)

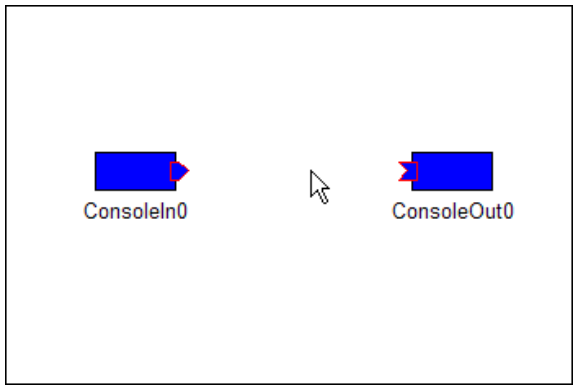

Fig. 4.18: Disconnecting components(b)

Moving connection lines If the line connecting 2 ports makes an elbow, it is possible to move it. A left click on such a line will make read markers appear on the segments of the line that can be moved. Dragging one of the red marker will make the line on which it lies move along. An horizontal line can be moved up and down while a vertical line can be moved right and left (Figure 4.19).

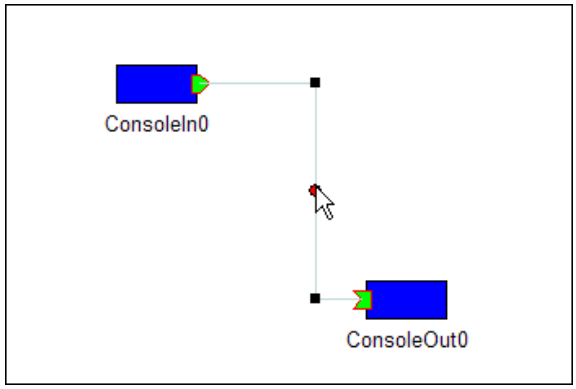

Fig. 4.19: Moving elbow lines

4.1.7.5 Starting and Stopping components

A right click on a component will cause a context menu to pop up (Figure 4.20).

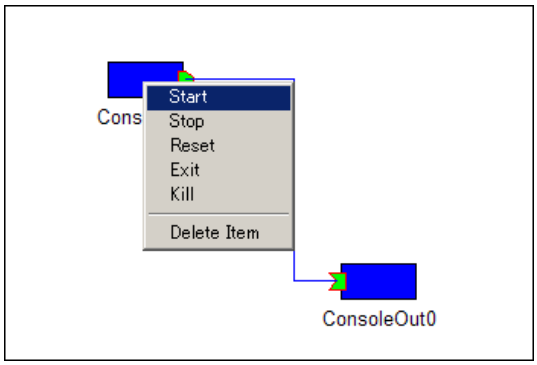

Fig. 4.20: Starting a component

The context menu contains the items listed below. From this menu, it is possible to control a component. It is possible to select Start or Stop, even if the the component is already respectively started or stopped. In that case, nothing will happen.

● Component Context Menu

Start	Start of component
Stop	Stop the component
Reset	Reset the component
Exit	Terminate the component
Kill	Kill the component
Delete Item	Delete the component

4.1.8 Saving and loading a system

Systems composed in RTCLink can be saved and reloaded.

4.1.8.1 Saving a System

There are 2 ways to save a system as described below.

Save System

Saves the system to an XML file using a default name (the title of the Sysem Composer Window). For example, if the title of the System Composer Window is [System:1], the file name will be System1.xml

● Saving a system using the default name

Select “File” - “Save System” in the menu bar
Right click in the System Composer Window and select “Save System” in the context menu

---

Save System as

Saves the system to an XML file using the name specified by the user. To do so,

● **Saving a system using a specific name**

Select “File” - “Save System As” in the menu bar
Right click in the System Composer Window and select “Save System As” in the context menu

---

In this case, a dialog wil open prompting for the name of the file (Figure 4.21).

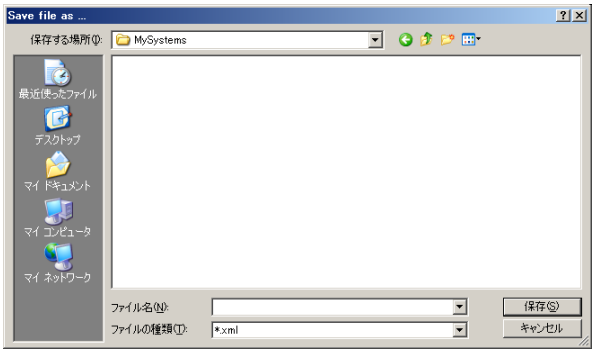

Fig. 4.21: ”File Save” dialog

The XML file will be saved after the user inputs the file name and click the OK button.

4.1.8.2 Loading a System

To load a system that has been saved as an XML file,

● **Loading a System**

Select “File” - “Open System” from the menu bar
Right click in the System Composer Window and select “Open System” in the context menu

---

A ”File Open” dialog appears (Figure 4.22).

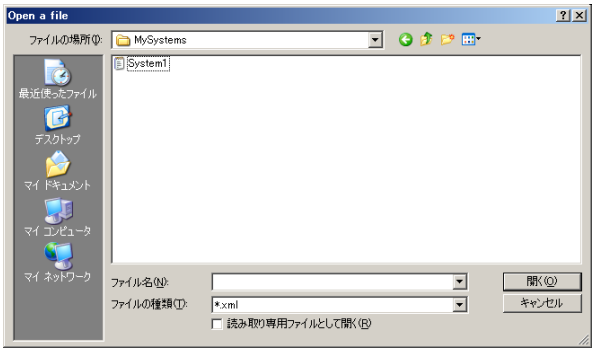

Fig. 4.22: "File Open" dialog

After the user input the file name and press the OK button, the system previously composed will be restored but the color of the component blocks will be white (Figure 4.23). At this stage, it is necessary to check if the system that was restored is still consistent or not.

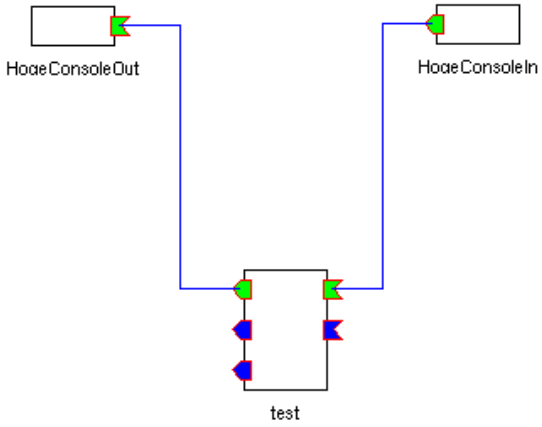

Fig. 4.23: Aspect of a system that has just been loaded

If, while checking the system, some component are drawn in black like in figure 4.24, it means that these component are not running. In that case, start the necessary components and reload the system.

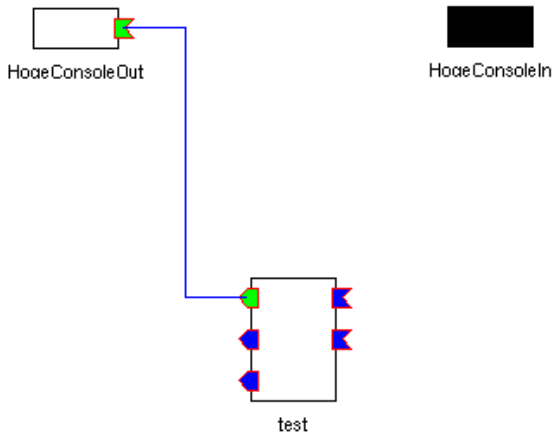

Fig. 4.24: Case where components are not running

Next, we will connect to the components. By right clicking in the background of the System Composer Window, we access the context menu and select “Connect”. The component blocks will then turn blue and it will be possible to operate them (like starting, stopping...)

● **Context menu when cheking a system**

Connect	Connect to components
Delete	Clear the current system

4.1.9 Refresh

For some reasons (like closing RTCLink while still connected or restarting components...) , RTCLink does not always reflect the actual state of the system and the connections between components. In this case, by right clicking in the background of the System Composer Window, we should access the context menu and select “Refresh”.

Doing a refresh will redraw the component blocks according to the actual state of the components and restore connections between components according to the system display in the System Composer Window. If connections between components that were not specified in the system Composer Window are detected, a dialog will prompt the user if the connection should be maintained or not.

4.2 rtc-template

rtc-template is a simple code generator that can produce the skeleton code of an RT-Component.

Presently, rtc-template supports the following functionalities.

● **Functionalities supported by rtc-template**

- Generation of skeleton of the module class inherited from RtcBase.
- Population of the component profile.
- Generation and registration of the InPorts.
- Generation and registration of the OutPorts.

The automatic generation of a custom interface based on the RTCBase interface is not yet supported. We are planning this feature for a future version.

Calling rtc-template - help will display the list of all supported options.

```
> rtc-template --help

Usage: rtc-template [OPTIONS]
Options:
    [--help]                                Print this help.
    [--c++]                                 Generate C++ template code.
    [--python]                              Generate Python template code.
    [--output[=output_file]]                Output base file name.
    [--module-name[=name]]                  Your module name.
    [--module-desc[=description]]           Module description.
    [--module-version[=version]]            Module version.
    [--module-author[=author]]              Module author.
    [--module-category[=category]]          Module category.
    [--module-comp-type[=component_type]] Component type.
    [--module-act-type[=activity_type]]     Component's activity type.
    [--module-max-inst[=max_instance]]      Number of maximum instance.
    [--module-lang[=language]]              Language.
    [--module-inport[=PortName:Type]]       InPort's name and tyoe.
    [--module-outport[=PortName:Type]]      OutPort's name and type
     :
```

4.2.1 --help

Displays the help section of rtc-template. The meaning and use of every supported options is described. Refer to it when necessary.

4.2.2 --c++

Generates the C++ code. If this option is specified, the following files will be generated (assuming here that the component module name - described hereafter - is "Foo") :

● **Files generated**

Foo.h	Header file for the Foo component
Foo.cpp	Source file for the Foo component
FooComp.cpp	Source file to execute the Foo compoent in standalone mode
Makefile.Foo	Makefile for the Foo component

4.2.3 --python

Generates the Python code If this option is specified, the following files will be generated (assuming here that the component module name - described hereafter - is "Foo") :

● **Files generated**

Foo,py	Python version of the Source file for the Foo component

4.2.4 --output

Specifies the name of the output files, ignoring the Component Module Name. If `--output=Bar` is specified, the following files will be generated.

● **Files generated**

Bar.h	Header file for the component
Bar.cpp	Source file for the component
BarComp.cpp	Source file to execute the compoent in standalone mode
Makefile.Bar	Makefile for the component

This option applies only if the `--c++` has been specified.

4.2.5 --module-name

Specifies the Module Name. The name specified will also be used as the class name. Specify a name that is compatible with the class naming rules in C++.

4.2.6 --module-desc

Specifies a short description of the component module. White spaces are allowed if the description is enclosed within double quotation marks.

4.2.7 --module-version

Specifies the version number.

4.2.8 --module-author

Specifies the component developer's name and affiliation.

4.2.9 --module-category

Specifies the category of the component.

4.2.10 --module-comp-type

Specifies the component type. Refer to section "Component Types" of 3.4 for a complete list of the possible values.

4.2.11 --module-act-type

Specifies the component activity type. Refer to section "Component Activity Types" of 3.4 for a complete list of the possible values.

4.2.12 --module-max-inst

Specifies the maximum number of simultaneously running instances.

4.2.13 --module-lang

Specifies the programming language used to develop the component. As this option is overridden by `--c++` or `--python`, it is not necessary to specify it.

4.2.14 --module-inport

Specifies an InPort supported by the component. This option can be specified several times to define the several InPorts a component may support.

● **Example of InPort Specification**

pos:TimedFloatSeq	Specifies an InPort named "pos" and supporting the TimeFloatSeq Data Type
vel:TimedFloat	Specifies an InPort named "vel" and supporting the TimeFloat Data Type
num:TimedShort	Specifies an InPort named "num" and supporting the TimeShort Data Type

The "Name" and "Type" part of this option are separated by a colon mark ":". Refer to section "Data Types" of 3.4 for a complete list of the supported data types.

4.2.15 --module-outport

Specifies an OutPort supported by the component. This option can be specified several times to define the several OutPorts a component may support. OutPorts are specified in the same way as InPorts.

4.3 rtm-config

`rtm-config` is a utility that stores all the settings necessary for building a component.

```
> ./rtm-config --help
Usage: rtm-config [OPTIONS]
Options:
        [--prefix[=DIR]]
        [--exec-prefix[=DIR]]
        [--version]
        [--libs]
        [--cflags]
        [--libdir]
        [--orb]
        [--idlc]
        [--idlflags]
```

4.3.1 --prefix

Stores the path to the directory where OpenRTM-aist has been installed.

```
> ./rtm-config --prefix
/usr/local
```

4.3.2 --exec-prefix

Stores the path to the directory where executable files have been installed.

```
> ./rtm-config --exec-prefix
/usr/local
```

The actual path to the directory is prefix+bin.

4.3.3 --version

Stores the version number of OpenRTM-aist.

```
> ./rtm-config --version
aist-0.2.0
```

As can be seen above, the format to store the version number is aist-x.x.x.

4.3.4 --libs

Stores the path to the directories where the linked libraries are installed. The format is the same as if it was directly passed to the linker. The data stored by rtm-config is used to specify the libraries in the Makefile. This allows for portable Makefiles.

```
> ./rtm-config --libs
-L/usr/local/lib -L/usr/local/lib -L/usr/local/lib -lpthread -lACE \
-lboost_regex -lomniORB4 -lomnithread -lomniDynamic4 -lRTC
```

4.3.5 --cflags

Stores the compile options. The format is the same as if it was directly passed to the compiler. The data stored by rtm-config is used to specify the compile options in the Makefile. This allows for portable Makefiles.

```
> ./rtm-config --cflags
-I/usr/local/include -I/usr/local/include -I/usr/local/include \
-I/usr/local/include -I/usr/local/include -I/usr/local/include/rtm/idl
```

4.3.6 --libdir

Stores the path to the directory where the various files in OpenRTM-aist should be installed.

```
> ./rtm-config --libdir
/usr/local/lib/OpenRTM
```

4.3.7 --orb

Stores the name of the ORB upon which OpenRTM-aist has been built. The example below shows that OpenRTM-aist was built upon OmniORB.

```
> ./rtm-config --orb
omniORB
```

4.3.8 --idlc

Stores the full path to the IDL compiler used to build OpenRTM-aist. This option must be specified if the user plans on defining new IDLs to support the components he will develop.

```
> ./rtm-config --idlc
/usr/local/bin/../bin/omniidl
```

4.3.9 --idlflags

Stores the options passed to the IDL compiler to build OpenRTM-aist.

```
> ./rtm-config --idlflags
-bcxx -Wba -nf
```

4.3.10 How to use rtm-config

Using `rtm-config` makes it easy to store all the machine specific information related to how OpenRTM-aist was built and installed. The compile and link option stored in `rtm-config` will mostly be used by Makefile. In the Makefile, a reference to `rtm-config -cflags` and `rtm-config -libs` as the compile and link options, will allow to use the same Makefile independently from the machine on which it is called.

4.4 rtm-naming

`rtm-naming` is a utility that hides the ORB specific options to run the naming service by providing a common wrapper. Presently, only omniORB is supported. However, as OpenRTM-aist will support other ORBs, starting their different naming service will be possible by using the same command options.

4.5 rtcd

`rtcd` is an executable generic component manager. After being started, any loadable module can be loaded into `rtcd`.

- 5- RTM Specification

OpenRTM-aist was developed by the National Institute of Advanced Industrial Science and Technology under the 21st century Robot Challenge Programme called “Development of the key enabling technologies for Robotics” sponsored by the New Energy and Industrial Technology Development Organization (NEDO). While the work toward the standardization of the interface definitions included in the RT-Middleware specification is still in progress, OpenRTM-aist is an open and free-to-use conforming implementation of the RT-Middleware.

Within this project which aimed at developing a middleware that would become a standard platform for the development of robotic systems, the RTM specification contains all the interface level standard specifications.

In this chapter, we will explain the interfaces specified in OpenRTM-aist and based on the RTM specifications, as well as their implementation. We will also, while reviewing the IDL interfaces specifications, examine the differences between the original interfaces included in RTM Specifications and the extensions provided by OpenRTM-aist.

5.1 OpenRTM-aist and the RTM Specification

RTM was a 3-year (2002 2004) joint research and development project between the National Institute of Advanced Industrial Science and Technology (AIST), Matsushita Electric Works, Ltd (MEW) and the Japan Robot Association (JARA) under the 21st century Robot Challenge Programme called “Development of the key enabling technologies for Robotics” sponsored by the New Energy and Industrial Technology Development Organization (NEDO), and aiming at the development of a middleware technology for networked robotic systems.

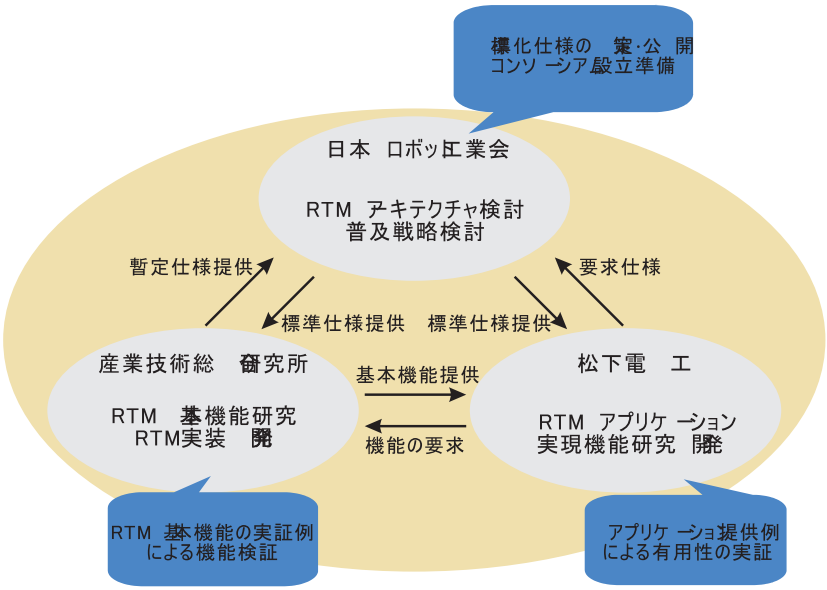

Fig. 5.1: Organization of the RTM project

The goal of the standardization activity within the RTM research and development has been to enable the interoperability between the components provided by diverse makers, component vendors, researchers, etc by providing a standard at the interface level. The final aim has been that two components supporting the same interface and placed in the same system could collaborate with each other and with other components independently from how they were implemented.

The present standard specification is open and anyone is free to provide his own conforming implementation. Moreover, the developer of such implementation is free to distribute or sell it under any type of licence. OpenRTM-aist is a free implementation of the RTM specification provided by the National Institute of Advanced Industrial Science and Technology. (Figure 5.2). Until now, OpenRTM-aist is the only free conforming implementation of the RTM interfaces specification. The RTM specification only defines the basic component interfaces, that is, the component object and InPort/OutPort basic interfaces. OpenRTM-aist provides a framework, as well as a set of tools and services, making easier the development of components compatible with the standard interface specifications.

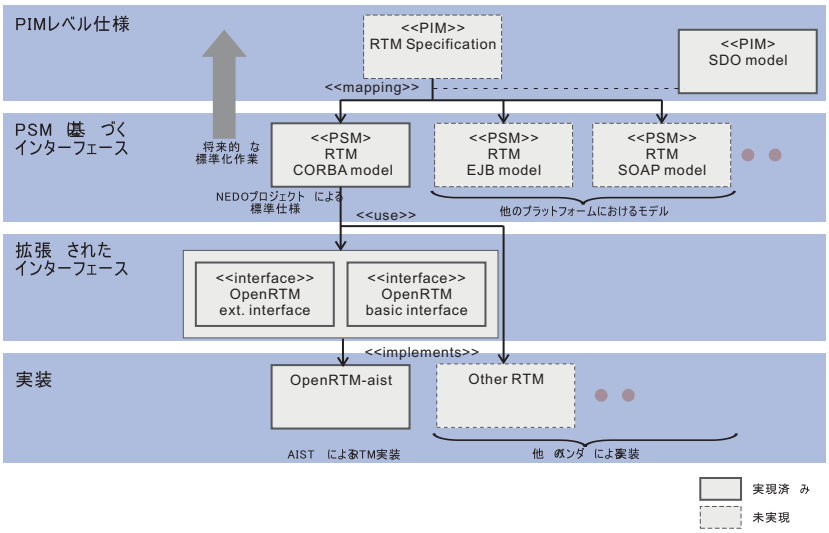

Fig. 5.2: Relation between the RTM Specification and OpenRTM-aist

Except for data types for InPort/OutPort, the manager, etc... which are necessary to execute a components, we tried not to include in the basic specifications any application dependent specification or parts that are not fundamental to ensure component interoperability. For example, the interfaces necessary to build a composite or a real-time component all derive from the basic `RTComponent` interfaces and are defined in the `RTCBase` interface.

5.2 Standardization of the RT-Middleware

We are presently working on having RTM recognized as an official OMG (Object Management Group) standard.

5.2.1 OMG (Object Management Group)

The OMG (Object Management Group), founded in 1989, is a not-for-profit consortium that produces and maintains computer industry specifications for interoperable applications based on the distributed object technology. It now is the biggest such organization and counts more than 800 members from around the world, including virtually every large company in the computer industry, and hundreds of smaller ones.

OMG's flagship specifications are the distributed object middleware CORBA (Common Object Request Broker Architecture) and UML (Unified Modeling Language) which specifies a uniform representation of object oriented software models [1].

[1] UML itself originally a modeling language developed by Grady Booch, James Rumbaugh ana Ivar Jacobson from Rational Software. However, it became an industry wide standard after its recognition by the OMG in Noverber 1997

5.2.2 MDA (Model Driven Architecture)

MDA (Model Driven Architecture) is the OMG's flagship specification.

In today's software technology, portability and interoperability are two major issues as the same software often needs to be reimplemented to support different platforms and these implementations often turn out not to be able to communicate properly.

MDA is an architecture supporting the developement of software systems and that emphasizes software modeling. With MDA, all software systems should first be specified by a model. From the model, a standard mapping to specific programming languages and platforms can automatically all the necessary code. MDA, by also addressing system management and integration issues, attemps to bring improvement in the efficiency of the the development of software systems.

As almost all software artifacts can be automatically derived from the model, the application of MDA may drastically reduce the cost of software development. Although the final idea of automatic code generation is very seducing, it as been seldom been put into practice so far. MDA receives now in the same criticisms as the case tools did in the 80's.

However, as the models dealt by MDA are fundamentally platform independent, they become therefore highly reusable and moreover provide an effective technique of communication between developers in charge of different parts of a system.

Using MDA, the developement of a system follows the 2 steps described below.

1. Creation of the PIM or "Platform Independent Model", a model that is independent form any OS, language or middleware technology.
2. From the PIM, Generation of the PSM or "Platform Specific Model", which is dependent on a specific OS, language and middleware technology.

5.2.3 PIM (Platform Independent Model)

The PIM is a model that is independent from any platform, that is, any OS, programming language and middleware technology. In the RTM project, the interface specifications took the form of IDL files, based on the CORBA technology. Therefore, the RTM specification is platform dependent and cannot be considered as a PIM. However, the ideas reflected in the interface definitions could be generalized and be standardized at the PIM level in the future. This standardization at the PIM level would let RTM to be applied and easily portedto other platforms such as JEB, DCOM, SOAP, XML-RPC, etc... (See figure 5.2).

5.2.4 PSM (Platform Specific Model)

The PSM is a model that relies on a specific platform. In case of the CORBA platform, the model is then expressed using IDL interfaces. The present RTM specification, by trying to set a standard at the CORBA IDL level, can be considered as a PSM (See figure 5.2).

5.3 RTM Specification Ver.0.1

5.3.1 RTMBase.idl

`RTMBase.idl` contains various definitions that are common to all parts of a system.

RTMBase.idl

```
module RTM {

  typedef short RtmRes;

  const RtmRes RTM_OK         = 0;
  const RtmRes RTM_ERR        = 1;
  const RtmRes RTM_WARNING    = 2;
  const RtmRes RTM_FATAL_ERR = 4;

  struct NamedValue {
    string name;
    any value;
  };

  typedef sequence<NamedValue> NVList;

  struct Time
  {
    unsigned long sec;    // sec
    unsigned long nsec;   // nano sec
  };
};
```

`RtmRes` is the type for values returned by RTM operations. An `RtmRes` typed return value is used to indicate if an operation executed normally. It can take the following values.

● **Operation return values**

`const RtmRes RTM_OK = 0;`	Normal completion
`const RtmRes RTM_ERR = 1;`	An error occured
`const RtmRes RTM_WARNING = 2;`	A warning occured
`const RtmRes RTM_FATAL_ERR = 4;`	A fatal error occured

`NamedValue` is defined as a structure that can contain any type of data. `NVList` is defined as a sequence of `Namedvalue`.

5.3.2 RTComponent

The following is the RTComponent IDL interface.

RTComponent.idl (1)

```
#include "RTMBase.idl"
#include "RTCInPort.idl"
#include "RTCOutPort.idl"

module RTM {
  interface RTComponent
  //    : NamedObject, PropertySet
  {
    readonly attribute string instance_id;
    readonly attribute string implementation_id;
    readonly attribute string description;
    readonly attribute string version;
    readonly attribute string maker;
    readonly attribute string category;

    typedef short ComponentState;

    const ComponentState RTC_UNKNOWN        = 0;
    const ComponentState RTC_BORN           = 1;
    const ComponentState RTC_INITIALIZING   = 2;
    const ComponentState RTC_READY          = 3;
    const ComponentState RTC_STARTING       = 4;
    const ComponentState RTC_ACTIVE         = 5;
    const ComponentState RTC_STOPPING       = 6;
    const ComponentState RTC_ABORTING       = 7;
    const ComponentState RTC_ERROR          = 8;
    const ComponentState RTC_FATAL_ERROR    = 9;
    const ComponentState RTC_EXITING        = 10;

    exception IllegalTransition {};
```

The `RTComponent` interface declares the main body of an RT-Component. All the RTM related interfaces are declared within the namespace called `RTM`.

Attributes of the profile

The following items are the information defined in the component profile.

● Information contained in the component profile

`readonly attribute string instance_id;`	Instance ID
`readonly attribute string implementation_id;`	Implementation ID
`readonly attribute string description;`	Short description
`readonly attribute string version;`	Version number
`readonly attribute string maker;`	Developer's name
`readonly attribute string category;`	Category

5.3.3 States of the component activity

`ComponentState` is the value type representing a state of the component activity and is equivalent to a CORBA short via the `typedef` statement. The following 11 states are defined

as constant values.

● **States of the Component Activity**

`const ComponentState RTC_UNKNOWN = 0;`	UNKNOWN state
`const ComponentState RTC_BORN = 1;`	BORN state
`const ComponentState RTC_INITIALIZING = 2;`	INITIALIZING state
`const ComponentState RTC_READY = 3;`	READY state
`const ComponentState RTC_STARTING = 4;`	STARTING state
`const ComponentState RTC_ACTIVE = 5;`	ACTIVE state
`const ComponentState RTC_STOPPING = 6;`	STOPPING state
`const ComponentState RTC_ABORTING = 7;`	ABORTING state
`const ComponentState RTC_ERROR = 8;`	ERROR state
`const ComponentState RTC_FATAL_ERROR = 9;`	FATAL_ERROR state
`const ComponentState RTC_EXITING = 10;`	EXITING state

The transition between these aforementioned states is depicted in figure 5.3.

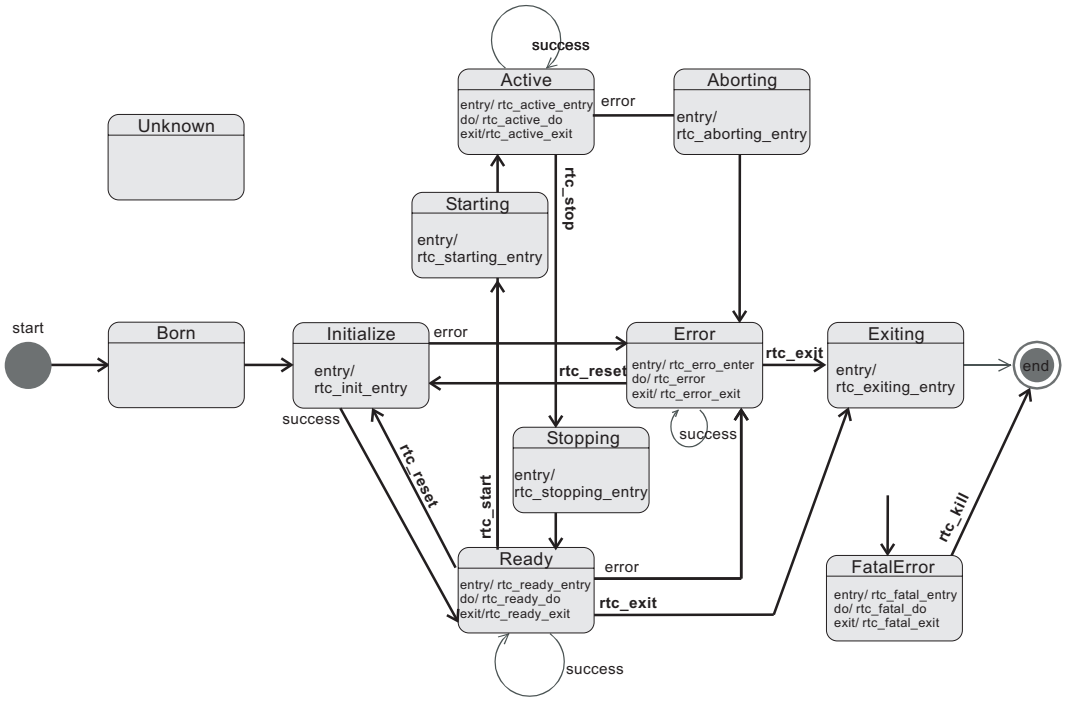

Fig. 5.3: RT-Component State Transitions

RTComponent.idl (2)

```
    RtmRes rtc_worker();

    RtmRes rtc_start() raises (IllegalTransition);
    RtmRes rtc_stop() raises (IllegalTransition);
    RtmRes rtc_reset() raises (IllegalTransition);
    RtmRes rtc_exit() raises (IllegalTransition);
    RtmRes rtc_kill();

    readonly attribute OutPort rtc_state;

    exception NoSuchName { string name; };
    readonly attribute InPortList inports;
    InPort get_inport(in string name) raises (NoSuchName);
    readonly attribute OutPortList outports;
    OutPort get_outport(in string name) raises (NoSuchName);

    //! Return connector port object reference list
    //    readonly attribute ConnectorList connectors;
    //    OutPort get_connector(in string name) raises (NoSuchName);

    //    readonly attribute PropertySet rtc_settings;

  };

  typedef sequence<RTComponent> RTComponentList;

};  // end of module RTM
```

`rtc_worker()` is the operation in which the main activity of the component is executed. Usually, this operation is executed internally and should not be called from an eternal entity. However, in the case of composite components, the external component in charge of the synchronization may have to call the operation directly to ensure the synchronization of the components.

`rtc_start()`, `rtc_stop()`, `rtc_reset()`, `rtc_exit()` and `rtc_kill)` are the operation that will initiate a state transition of component. The state transitions of the component activity follow the state machine described in figure 5.3.

If an operation is called and which is incoherent with the state in which the component is at that time, an `IllegalTransition` exception is raised.

`rtc_state` returns the reference to the OutPort which outputs the state of the component. By using this OutPort, it is possible to acquire the state of the component by either pull or a push style data transfer.

● Component Activity and State Transiton Operations

`RtmRes rtc_worker();`	Main body of the Activity
`RtmRes rtc_start() raises (IllegalTransition);`	Transition to the ACTIVE State
`RtmRes rtc_stop() raises (IllegalTransition);`	Transition to the READY State
`RtmRes rtc_reset() raises (IllegalTransition);`	Transition to the INITIALIZING State
`RtmRes rtc_exit() raises (IllegalTransition);`	Transition to the EXITING State
`RtmRes rtc_kill();`	Transition to the EXITING State after a FATAL_ERROR occured
`readonly attribute OutPort rtc_state;`	OutPort publishing the Component State

A component can own one or more data Input / Output objects called respectively InPort and OutPort. By using the operations described below, one can obtain their object reference.

● **InPort/OutPort**

`exception NoSuchName { string name;};`	The Port with the specified name does not exist
`readonly attribute InPortList inports;`	Obtain the list of available InPorts
`InPort get_inport(in string name) raises (NoSuchName);`	Obtain the InPort with the specified name
`readonly attribute OutPortList outports;`	Obtain the list of available OutPorts
`OutPort get_outport(in string name) raises (NoSuchName);`	Obtain the OutPort with the specified name

Finally, `RTComponentList` contains a sequence of object references to RT-Components. It is used to simultaneously handle several components.

5.3.4 RTCInPort.idl

RTCInPort.idl

```
#include "RTMBase.idl"

module RTM {
  typedef string SubscriptionID;

  struct PortProfile
  {
    string name;
    CORBA::TypeCode port_type;
    NVList properties;
  };

  interface InPort
  {
    exception Disconnected{};

    void put(in any data) raises(Disconnected);

    readonly attribute PortProfile profile;
  };

  typedef sequence<InPort> InPortList;
  typedef sequence<PortProfile> PortProfileList;

}; // end of module RTM
```

First, `SubscriptionID` is declared as a CORBA string. The `SubscriptionID` is used as an subscription identifier when subscribing to an OutPort. It usually contains a UUID.

The `PortProfile` structure contains the information related to the Inport/OutPort profile. At the top level are the name of the port : `name` and its data type : `port_type`. Other information are stored in a sequence of `NamedValue` : `NVList`.

Data can be sent to the InPort by using the `put()` method declared in its interface.. If an OutPort that has been disconnected tries to call the `put()` method, the `Disconnected` will be raised. Information concerning the profile of an Inport are accessible from the `profile` member.

5.3.5 RTCOutPort.idl

RTCOutPort.idl

```
#include "RTMBase.idl"
#include "RTCInPort.idl"

module RTM {
  typedef short SubscriptionType;

  const SubscriptionType OPS_ONCE                = 0;
  const SubscriptionType OPS_PERIODIC            = 1;
  const SubscriptionType OPS_NEW                 = 2;
  const SubscriptionType OPS_TRIGGERED           = 3;
  const SubscriptionType OSP_PERIODIC_NEW        = 4;
  const SubscriptionType OPS_NEW_PERIODIC        = 5;
  const SubscriptionType OPS_PERIODIC_TRIGGERED  = 6;
  const SubscriptionType OPS_TRIGGERRED_PERIODIC = 7;

  struct SubscriberProfile
  {
    SubscriptionType subscription_type;
    boolean event_base;
    NVList properties;
  };

  interface OutPort
  {
    any get();
    RtmRes subscribe(in InPort in_port, out SubscriptionID id,
                     in SubscriberProfile profile);
    RtmRes unsubscribe(in SubscriptionID id);
    readonly attribute InPortList inports;
    readonly attribute PortProfile profile;
  };

  typedef sequence<OutPort> OutPortList;
}; // end of module RTM
```

First, the type of subscription to an OutPort, `SubscriptionType`, is defined as a CORBA short by using a `typedef`.

The following constants declared the subscription types that are supported.

● **Subscription Types**

OPS_ONCE	= 0;	Request to send the latest present data
OPS_PERIODIC	= 1;	Request to send data periodically
OPS_NEW	= 2;	Request to send every new data
OPS_TRIGGERED	= 3;	Request to send every data that meets the condition d
OSP_PERIODIC_NEW	= 4;	Request to send data periodically but only if there is a
OPS_NEW_PERIODIC	= 5;	Request to send every new data but not faster than th
OPS_PERIODIC_TRIGGERED	= 6;	Request to send data periodically but only if the data
OPS_TRIGGERRED_PERIODIC	= 7;	Request to send every data that meets the condition d

With `OPS_ONCE`, the data is sent only once upon subscription. The OutPort and InPort are then disconnected.

With `OPS_PERIODIC`, the data is sent to the subscriber at the rate specified during the subscription.

With `OPS_NEW`, the data is sent to the subscriber as soon as it has been updated in the OutPort.

with OPS_TRIGGERED, the data is sent only when the it meets the condition defined by the trigger. Conditions defined by a trigger applies to a port (not just to a subscription). For example, such trigger could be defined as "Send only if the value of the data in the OutPort is superior or equal to 4.0".

With OPS_PERIODIC_NEW, the data is sent to the subscriber at the rate specified during the subscription but only if it has been updated in the OutPort. If the sending period was set to Δt [s], the updated data will not be sent before the next occurrence of Δt [s].

With OPS_NEW_PERIODIC, the data is sent to the subscriber as soon as it has been updated in the OutPort but not faster than the specified rate.

OPS_PERIODIC_TRIGGERED works in the same way as OPS_PERIODIC_NEW does, except that the condition to send the data is not that it has simply been updated, but that it meets the conditions defined by the trigger.

OPS_TRIGGERED_PERIODIC works in the same way as OPS_NEW_PERIODIC does, except that the condition to send the data is not that it has simply been updated, but that it meets the conditions defined by the trigger.

The SubscriptionType defining the type of subscription is part of the following SubscriberProfile structure.

```
struct SubscriberProfile
{
  SubscriptionType subscription_type;
  boolean event_base;
  NVList properties;
};
```

The SubscriptionType described above is specified by SubscriptionType subscription_type. event_base is a CORBA Boolean that specifies if the data concerned with subscription should be sent via the Event/Notification Service or not. Other properties related to a subscription are stored in NVList properties.

To subscribe to an OutPort, the members of this structure must be initialized adequately and passed over when calling the subscribe() operation on the OutPort.

Then comes the OutPort interface.

```
interface OutPort
{
  any get();
  RtmRes subscribe(in InPort in_port, out SubscriptionID id,
                   in SubscriberProfile profile);
  RtmRes unsubscribe(in SubscriptionID id);
  readonly attribute InPortList inports;
  readonly attribute PortProfile profile;
};

typedef sequence<OutPort> OutPortList;
```

The current value of an OutPort can be acquired by calling its `get()` operation.

Subscribing to an OutPort by calling its `subscribe()` operation will cause it to send its data periodically in a push-like manner. The object reference of the subscribing InPort along with a subscriber profile structure should be passed over as argument. If the subscription succeeds, `subscribe()` will return `RTM_OK` and intialize the output argument `SubscriptionID id` with the UUID (Universally Unique IDentifier) assigned to the subscription.

The subscription can then be discontinued by calling the `unsubscribe()` operation with this UUID as argument.

The attribute `InPortList inports` holds a list of all the InPort that have subscribed to the OutPort. The profile of the OutPort itself is stored in the attribute `PortProfile profile`.

5.4 Interfaces specified in OpenRTM-aist

OpenRTM-aist, while while remaining compatible with the RTM Specification, provides several extensions and has added the definition of new object interfaces.

● IDL Files for the Interfaces defined in OpenRTM-aist

`RTCBase.idl`	Specifies the extensions to the RTComponent interface
`RTCProfile.idl`	Specifies the extensions to the structure that stores an RT-Component profile
`RTCDataType.idl`	Specifies standard data types supported by the InPort and OutPort
`RTCManager.idl`	Specifies the interface of the manager that manages the lifecycle of an RT-Component

5.4.1 RTCBase.idl

OpenRTM-aist provides `RTCBase` as an extension to the definition of an RT-Component. This extended interface mostly defines the extra attributes and operations necessary to facilitate the creation of composite components.

RTCBase.idl

```
#include "RTComponent.idl"
#include "RTCProfile.idl"

module RTM {

  interface RTCBase;
  typedef sequence<RTCBase> RTCBaseList;

  interface RTCBase
    : RTComponent
  {
    RtmRes rtc_ready_entry();
    RtmRes rtc_ready_do();
    RtmRes rtc_ready_exit();
    RtmRes rtc_active_entry();
    RtmRes rtc_active_do();
    RtmRes rtc_active_exit();
    RtmRes rtc_error_entry();
    RtmRes rtc_error_do();
    RtmRes rtc_error_exit();
    RtmRes rtc_fatal_entry();
    RtmRes rtc_fatal_do();
    RtmRes rtc_fatal_exit();
    RtmRes rtc_init_entry();
    RtmRes rtc_starting_entry();
    RtmRes rtc_stopping_entry();
    RtmRes rtc_aborting_entry();
    RtmRes rtc_exiting_entry();

    RtmRes rtc_stop_thread();
    RtmRes rtc_start_thread();

    RtmRes rtc_set_parent(in RTCBase comp);
    RtmRes rtc_add_component(in RTCBase comp);
    RtmRes rtc_delete_component(in RTCBase comp);
    RtmRes rtc_replace_component(in RTCBase comp1, in RTCBase comp2);
    RtmRes rtc_replace_component_by_name(in string name1, in string name2);
    RTCBaseList rtc_components();
    RTCBase rtc_get_component(in string name);

    RtmRes rtc_attach_inport(in InPort in_port);
    RtmRes rtc_attach_inport_by_name(in RTCBase comp, in string name);
    RtmRes rtc_detatch_inport(in InPort in_port);
    RtmRes rtc_detatch_inport_by_name(in string name);
```

```
    RtmRes rtc_attach_outport(in OutPort out_port);
    RtmRes rtc_attach_outport_by_name(in RTCBase comp, in string name);
    RtmRes rtc_detatch_outport(in OutPort out_port);
    RtmRes rtc_detatch_outport_by_name(in string name);

    readonly attribute RTCProfile profile;
  };
};
```

The first part defines the CORBA operations equivalent to all the state methods of the component activity.

アクティビティオペレーション

```
    RtmRes rtc_ready_entry();
    RtmRes rtc_ready_do();
    RtmRes rtc_ready_exit();
    RtmRes rtc_active_entry();
    RtmRes rtc_active_do();
    RtmRes rtc_active_exit();
    RtmRes rtc_error_entry();
    RtmRes rtc_error_do();
    RtmRes rtc_error_exit();
    RtmRes rtc_fatal_entry();
    RtmRes rtc_fatal_do();
    RtmRes rtc_fatal_exit();
    RtmRes rtc_init_entry();
    RtmRes rtc_starting_entry();
    RtmRes rtc_stopping_entry();
    RtmRes rtc_aborting_entry();
    RtmRes rtc_exiting_entry();
```

Having these operations declared at the interface level enables to externalize the process (thread) that executes the activity to another component. The goal is to be able to create a composite component that will execute the activity of a group of several components either synchronously or sequentially.

Operations to control the Activity Thread

```
    RtmRes rtc_stop_thread();
    RtmRes rtc_start_thread();
```

These operations control the execution of the thread which execute the component internal activity. For a composite component, in order externalize the process (thread) that executes the activity of components, it is first necessary to stop their internal thread. This is in order to be able to control the component activity thread from outside the component that these operation had to be defined at the interface level.

Operations controlling the parent/child relation between components

```
RtmRes rtc_set_parent(in RTCBase comp);
RtmRes rtc_add_component(in RTCBase comp);
RtmRes rtc_delete_component(in RTCBase comp);
RtmRes rtc_replace_component(in RTCBase comp1, in RTCBase comp2);
RtmRes rtc_replace_component_by_name(in string name1, in string name2);
RTCBaseList rtc_components();
RTCBase rtc_get_component(in string name);
```

These are the operations used to control the parent/child relationship between components. In a usual single bodied component, the `rtc_set_parent()` has no effect and returns `RTM_ERR`. However, in the case of a composite component, these operations allows for assigning several children component to a parent component.

The Inport Attach Operations

```
RtmRes rtc_attach_inport(in InPort in_port);
RtmRes rtc_attach_inport_by_name(in RTCBase comp, in string name);
RtmRes rtc_detatch_inport(in InPort in_port);
RtmRes rtc_detatch_inport_by_name(in string name);
```

Using these operations, a composite component owning children components can expose their InPorts as if they were its own InPorts. Practically, the parent component only stores the object references of the Inports of its children components.

The OutPort Attach Operations

```
RtmRes rtc_attach_outport(in OutPort out_port);
RtmRes rtc_attach_outport_by_name(in RTCBase comp, in string name);
RtmRes rtc_detatch_outport(in OutPort out_port);
RtmRes rtc_detatch_outport_by_name(in string name);
```

Using these operations, a composite component owning children components can expose their OutPorts as if they were its own OutPorts. Practically, the parent component only stores the object references of the OutPorts of its children components.

5.4.2 RTCProfile.idl

`RTCProfile.idl` contains the definition of the RTCProfile interface. `RTCProfile` is a structure that contains the set of properties (like `instance_id, implementation_id`, etc...) defined in the `RTComponent` interfaces, as well as some extra properties. One of the extension of `RTCBase` provides way to get the complete profile in a single operation.

RTCProfile.idl

```
module RTM {
  interface RTComponent;

  enum RTComponentType {
    STATIC,
    UNIQUE,
    COMMUTATIVE
  };

  enum RTCActivityType {
    PERIODIC,
    SPORADIC,
    EVENT_DRIVEN
  };

  enum RTCLangType {
    COMPILE,
    SCRIPT
  };

  struct RTCProfile
  {
    string name;
    string instance_id;
    string implementation_id;
    string description;
    string version;
    string maker;
    string category;
    RTComponentType component_type;
    RTCActivityType activity_type;
    long max_instance;
    string language;
    RTCLangType language_type;
    string module_profile_file;
    PortProfileList outport_profile_list;
    PortProfileList inport_profile_list;
  };
};
```

The properties defined below (like `instance_id, implementation_id`, etc...) have exactly the same semantics and values as the one defined in the `RTComponent` interface.

● **Basic Profile**

```
string instance_id;
string implementation_id;
string description;
string version;
string maker;
string category;
```

`RTComponentType` specifies the type of the component and how the component should be instantiated. A component can be of the following types.

● `RTComponentType`

STATIC	The component is instantiated once when registered to the manager. The instantiation of the component is restricted to one. This type is particularly suited to component directly dealing with hardware.
UNIQUE	Dynamic instantiation and destruction of the component is possible. However, As component0 and component1 may have specific internal states, they cannot be used interchangeably.
COMMUTATIVE	These components can be used interchangeably. Thi type of component is especially suited for raw software logic.

`RTCActivityType` specifies the type of activity of a component. The following types are supported.

● `RTCActivityType`

PERIODIC	The main activity is realized within a fixed-time periodic thread. This periodic thread can become a real-time periodic thread on top of a real-time OS such as ART-LINUX.
SPORADIC	The main activity is also realized by a periodic thread but its cycle time is not fixed.
EVENT_DRIVEN	The component only reacts to requests from external entities, like calls to its CORBA interface from other components.

Other properties are listed below.

● **Other properties**

`long max_instance;`	Maximum number of instances
`string language;`	Programming language used to develop the component
`RTCLangType language_type;`	Type of programming language used to develop the component
`string module_profile_file;`	Name of the file where the Module Profile is stored

`RTCProfile` also stores the list of the profile of all InPort and OutPort provided by a component.

● **List of InPort and OutPort Profiles**

`PortProfileList outport_profile_list;`	List of InPort Profiles
`PortProfileList inport_profile_list;`	List of OutPort Profiles

`PortProfileList` is a sequence of `PortProfile` defined in the file `RTCInport.idl`.

5.4.3 RTCDataType.idl

The file `RTCDataType.idl` is as follows.

RTCDataType.idl (Basic Data Types)

```
#include "RTMBase.idl"

module RTM {
  struct TimedState
  {
    Time tm;
    short data;
  };
 struct TimedShort
  {
    Time tm;
    short data;
  };
 struct TimedLong
  {
    Time tm;
    long data;
  };
  struct TimedUShort
  {
    Time tm;
    unsigned short data;
  };
  struct TimedULong
  {
    Time tm;
    unsigned long data;
  };
  struct TimedFloat
  {
    Time tm;
    float data;
  };
  struct TimedDouble
  {
    Time tm;
    double data;
  };
  struct TimedChar
  {
    Time tm;
    char data;
  };
  struct TimedBoolean
  {
    Time tm;
    boolean data;
  };
  struct TimedOctet
  {
    Time tm;
    octet data;
  };
  struct TimedString
  {
    Time tm;
    string data;
  };
```

OpenRTM-aist provides the definition of the data types that can be exchanged between InPorts and OutPort. These data types all come with a timestamp and either contains a CORBA basic data type or a sequence of CORBA basic data type.

RTCDataType.idl (Sequence Types)

```
  struct TimedShortSeq
  {
    Time tm;
    sequence<short> data;
```

```
  };
  struct TimedLongSeq
  {
    Time tm;
    sequence<long> data;
  };
  struct TimedUShortSeq
  {
    Time tm;
    sequence<unsigned short> data;
  };
  struct TimedULongSeq
  {
    Time tm;
    sequence<unsigned long> data;
  };
  struct TimedFloatSeq
  {
    Time tm;
    sequence<float> data;
  };
  struct TimedDoubleSeq
  {
    Time tm;
    sequence<double> data;
  };
  struct TimedCharSeq
  {
    Time tm;
    sequence<char> data;
  };
  struct TimedBooleanSeq
  {
    Time tm;
    sequence<boolean> data;
  };
  struct TimedOctetSeq
  {
    Time tm;
    sequence<octet> data;
  };
  struct TimedStringSeq
  {
    Time tm;
    sequence<string> data;
  };
};
```

5.4.4 RTCManager.idl

The `RTCManager` is the object which loads and instantiate the `RTComponent` modules and manage their lifecycle.

RTCProfile.idl

```
#include "RTMBase.idl"
#include "RTComponent.idl"
#include "RTCBase.idl"

module RTM
{
  typedef sequence<string> ComponentFactoryList;

  interface RTCManager
  {
    RtmRes load(in string pathname, in string initfunc);
    RtmRes unload(in string pathname);
    RTCBase create_component(in string comp_name,
                             out string instance_name);
    RtmRes delete_component(in string instance_name);
    ComponentFactoryList component_factory_list();
    RTCBaseList component_list();
    RtmRes command(in string cmd, out string ret);
  };

}; // end of namespace RTM
```

The `load()` and `unload()` operations are defined and will respectively dynamically load and unload modules.

● **Operation for loading and unloading modules**

`load(in string pathname, in string initfunc);`	Loads a module
`unload(in string pathname);`	Unload a module

The `create_component()` and `delete_component()` operations respectively create and delete a component.

● **Operations for creating and deleting a component**

`create_component(in string comp_name, out string instance_name);`	Create a component
`delete_component(in string instance_name);`	Delete a component

The remaining operations include `component_factory_list()`, which returns the list of all the component factories, and `component_list()`, which returns the list of all the instances of components.

● **Other Operations**

`component_factory_list();`	Return the list of component factories
`component_list();`	Return the list of all instances of components
`command(in string cmd, out string ret);`	Simple command interpreter

5.5 Extension and standardization of OpenRTM-aist

As of today, the RTM specification are on their way to standardization. It is therefore likely that the interface specifications may be altered in the future. One goal of the standardization process is to combine the feedback provided by as many real-world application as possible so as to provide specification that are even easier to use.

It is also likely that the extensions provided by OpenRTM-aist will evolve in the future and that some features may be added, combined or simply removed.

Appendix A Building the dependency packages

A.1 Building ACE

A.1.1 ACE (The ADAPTIVE Communication Environment)

The ADAPTIVE Communication Environment (ACE) is a freely available, open-source object-oriented (OO) framework that implements many core patterns for concurrent communication software. Its main features are :

- Increased portability
- Increased software quality based on design patterns
- Increased efficiency and predictability
- Easier transition to standard higher-level middleware (such as CORBA)

ACE provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of OS platforms. The communication software tasks provided by ACE include event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, concurrent execution and synchronization. OpenRTM-aist has been implemented using most of the functionalities provides by ACE. ACE supports the following platforms :

● **Platforms supported by ACE**

Windows (WinNT 3.5.x, 4.x, 2000, Embedded NT, XP, Win95/98, WinCE) as well as most development environments (MSVC++, Borland C++ Builder, IBM Visual Age (32-, 64-bit Intel/Alpha) Mac OS X, Solaris 1.x, 2.x (SPARC / Intel), SGI IRIX 5.x, 6.x, DG/UX, HP-UX 9.x, 10.x, 11.x, Tru64UNIX 3.x, 4.x, AIX 3.x, 4.x, 5.x, UnixWare, SCO, Debian Linux 2.x, RedHat Linux 5.2, 6.x, 7.x, 8x, and 9.x, Timesys Linux, FreeBSD, NetBSD, LynxOS, VxWorks, ChorusOS, QnX Neutrino, RTEMS, OS9, PSoS, OpenVMS, MVS OpenEdition, CRAY UNICOS

A.1.2 Downloading ACE

The ACE, as well as TAO, the ORB based on ACE, distributions are available on the following URLs

ACE `http://www.cs.wustl.edu/ schmidt/ACE.html`

TAO `http://www.cs.wustl.edu/ schmidt/TAO.html`

They can be downloaded from :

`http://deuce.doc.wustl.edu/Download.html`

The distribution is packaged into a compressed tar.gz, tar.bz or zip file.

A.1.3 Building ACE on a UNIX-like OS

To build ACE, follow the instructions provided in the INSTALL file in the `ACE_wrapper` directory. The procedure is as follow.

- Set the environment variable ACE_ROOT
- Edit the platform.macros.GNU file
- Edit the config.h file
- Build ACE

A.1.4 Setting the ACE_ROOT environement variable

We first have to set the environement variable called ACE_ROOT

[Case using csh]
```
$ setenv ACE_ROOT [Path to the directory where ACE was installed]
```
[Case using bash]
```
$ ACE_ROOT=[Path to the directory where ACE was installed]
$ export ACE_ROOT
```

A.1.4.1 Editing the platform.macros.GNU file

From now on, we will assume the `ACE_ROOT` has been set to `ACE_wrapper`, the directory where ACE was decompressed. The directory called `$ACE_ROOT/include/makeinclude/` contains the macros adapted to each supported platform and used by Makefile. Here, we will assume that ACE will be built using gcc(g++) in a Linux environment. The macro that supports gcc(g++) on Linux is `platform_linux.GNU`. We will then create the symbolic link platform.macros.GNU that points to this file.

```
> cd $ACE_ROOT/include/makeinclude/
> ln -s platform_linux.GNU platform.macros.GNU
```

A.1.4.2 Setting the config.h file

Next, we have to set the config.h file. The directory called `$ACE_ROOT/ace/` contains the config.h files adapted to each supported platform. Assuming that ACE will be built on Linux, we will create the symbolic link config.h that points to config-linux.h.

```
> cd $ACE_ROOT/ace/
> ln -s config-linux.h config.h
```

A.1.5 Building ACE

ACE is now ready to be built. After changing the current directory to `$ACE_ROOT`, we will execute `make`.

```
> cd $ACE_ROOT
> make
```

ACE being a rather big system, it may take time to build it.

A.2 Building boost

A.2.1 The boost C++ Library

boost is a collection of C++ template libraries. While STL already provides a collection of standard C++ template libraries, boost aims at providing an even richer set of standard template libraries gathered by the members of a standardization committee. In the future, boost may therefore become the standard template library for C++.

As of today, as it is not yet a standard, few platforms support it by default. In that case, one must build and install it by himself.

As boost is mostly a library of templates, there is,in most of the case, no actual need for builing it and just installing the header files should be enough. However, as OpenRTM--aist relies on the Regular Expression library which requires to be built, it will therefore be necessary to build boost. We will now explain how to do it step by step.

Now, if boost is already included with your system, you can as well use it as is.

A.2.2 Downloading boost

To download boost, go to `http://www.boost.org` and follow the menu "Download".

Once downloaded, decompress the package.

```
> tar vxzf boost_1_32_0.tar.gz
```

A.2.3 Building bjam

First, we will built `bjam`, a tool necessary to build boost. Running the `build.sh` script provided with the distribution will build bjam. Usually, the script will detect the environment and build bjam automatically.

```
> cd boost_1_32_0
> cd tools/build/jam_src
> ./build.sh
 :  (will start the building process)
> ls bin.(system_name)
> ls bin.linuxx86 :  for example
> ls
bjam* jam* mkjambase* yyacc*
> cd - (return to the boost top directory)
```

The binary file resulting from the build will be placed in the directory called `bin.(system_name)`

A.2.4 Building boost

First, in order to build the Python extension library of boost, we must set `PYTHON_VERSION` defined in the file `tools/build/v1/python.jam` to the version of Python we will be using. The default version is Python2.2.

For more details about the other settings, please refer to the documentation provided with boost.

Using `bjam`, we will now built boost as follow.

```
> tools/build/jam_src/bin.(system_name)/bjam -sTOOLS=gcc
 --prefix=/usr/local -sPYTHON_ROOT=/usr -sPYTHON_VERSION=2.3
 :
The building process starts
 :
```

From here, we will build boost assuming that it has been copied into the a directory called `/urs/local/[include|lib]` and that Python version 2.3 has been installed in `/usr`. For more details about the build options please refer to the documentation provided with boost.

A.2.5 Installing boost

Finally, after becoming root, we can install boost.

```
> su # tools/build/jam_src/bin.(system_name)/bjam -sTOOLS=gcc
 --prefix=/usr/local -sPYTHON_ROOT=/usr -sPYTHON_VERSION=2.3 install
 :
The installation starts
 :
```

Upon installation, the filename of the shared libraries in boost may be decorated with the version number as well as the compiler name (like `libboost_regex-gcc-1_32.so.1.32.0` or `libboost_regex-gcc-1_32.so`). In that case, it may be necessary to create a symbolic link.

```
# cd /usr/local/lib
# ln -s libboost_regex-gcc-1_32.so.1.32.0 libboost_regex.so
```

In any case, when linking OpenRTM, the file `libboost_regex.so` must be found under `/usr/local/lib` or `/usr/lib`.

A.3 Building omniORB

A.3.1 omniORB

OmniORB is a free ORB (Object Request Broker) implementation conforming with the version 2.6 of CORBA (Common Object Request Broker Architecture). First developed by AT&T Lab., the project migrated to sourceforge after the AT&T research center ceased its activity. Compared to other freely available CORBA implementations such as TAO or MICO, omniORB supports fewer of the CORBA service. This is however compensated by its high performance and speed.

A.3.2 Downloading omniORB

omniORB is available for download from its sourceforge website `http://omniorb.sourceforge.net/` Beware that a google search for omniORB may sometimes return a link to the old project page at AT&T Lab.

Once downloaded, the package should be decompressed in an appropriate directory.

```
> tar vxzf omniORB-4.0.5.tar.gz
> cd omniORB-4.0.5
```

A.3.3 Building omniORB

We will build omniORB following the instructions included in its documentation.

The basic procedure is described below. Basically, as omniORB makes use of autoconf, automake, it can be built simply by executing `./configure` and `make`.

We need to make the `build` directory for `configure` to save temporary install and object files. These are the instructions written in the manual.

```
> tar vxzf omniORB-4.0.5.tar.gz
> cd omniORB-4.0.5
> mkdir build
> ../configure [options]
```

Below is the list of the main options supported by `configure`.

Options for `configure`

`--prefix=dir`	Specifies the target directory
`--disable-static`	Does not build the static libraries
`--enable-thread-tracing`	Setting this option will facilitate the debugging of thread spawned by omniORB (set by default). As it incurs a slight performance drop, one may want to disable this option and set `--disable-thread-tracing` instead.
`--with-openssl=dir`	Specifies the directory where openSSL has been installed
`--with-omniORB-config=dir/file`	Specifies the path to the omniORB configuration file. The default value is `/etc/omniORB.cfg`.
`--with-omniNames-logdir=dir/file`	Speficied the pathe to the omniORB log file. The default value is `/var/omninames`.

A.3.4 Installing omniORB

We finally will install omniORB by become root and executing `make install` under the `build` directory.

```
> su
# make install
```