Run-time management of component-based robot software from a command line

Geoffrey Biggs, Noriaki Ando, and Tetsuo Kotoku

Intelligent Systems Research Institute
National Institute of Advanced Industrial Science and Technology (AIST)
AIST Tsukuba Central 2, Tsukuba, Ibaraki 305-8568, Japan

Abstract. Component-based software is a major recent design trend in robotics. It brings many benefits to system design, implementation and maintenance. The management of such systems often depends on graphical tools. These tools are powerful and provide a rapid way to layout component networks. However, they also typically require considerable resources to run. This is not ideal in robotics, where low-resource environments are common. We have created a set of command-line tools for use with the OpenRTM-aist component-based robot middleware. The tools follow the UNIX philosophy of simplicity and aggregation. These tools allow whole component-based systems to be created, managed and monitored from a command-line. They are ideal for use in environments where a graphical interface is not available. By combining tools together, more complex functionality can be easily created.

1 Introduction

Component-based software design and implementation is a current trend in software engineering. Software is divided into individual components, each with a well-defined interface that specifies what functionality that component provides. Multiple software components are combined together into a complete software system in much the same way as the well-defined hardware components of electrical circuits are combined to create a complete hardware system [12].

Component-based practices bring many benefits to software design, implementation, maintenance and reuse, including known interfaces that act as "contracts" between components, "separation of concerns" (each component only deals with its individual problem), isolation testing, and rapid development of new systems using existing commoditised software resources.

These benefits also apply to the design, implementation, maintenance and reuse of robot software. For example, the componentisation of hardware drivers and algorithms allows robot systems to be built from pre-existing, ideally off-the-shelf, software components. As a result, component-based software is a major trend in robotics, particularly service robotics. Recent examples include OpenRTM-aist [1], ORCA [2], ROS [8] and OPRoS [11].

A key part of using component-based software is interacting with the component network that makes up the software system, both when designing the system and when monitoring and maintaining the system in its running state. There are a variety of methods available for this. Interaction may be via a specialised tool designed to monitor a specific network of components. Alternatively, a generic tool for the component architecture upon which the software system is built may be used. In some cases, no such tool may be available, with all interaction taking place through configuration and log files.

The work in this paper presents a set of tools designed for managing systems using the OpenRTM-aist component-based framework. These tools differ from the usual approach in that they apply a file-system abstraction to the components for interaction, they are command-line tools and they follow the UNIX philosophy of being small and using aggregation to add power. Such a collection of simple tools gives great flexibility and ease of use to developers using component-based robotics software, particularly when the interaction method may be constrained by unique environmental factors.

The following section describes the framework that is the focus of this work. Section 3 discusses the background for the work. Section 4 discusses the tools themselves. Discussions and conclusions are given in Sections 5 and 6.

2 OpenRTM-aist

OpenRTM-aist [1] is a component-based architecture for intelligent systems, known in Japan as "Robot Technology," or "RT." OpenRTM-aist implements the Robot Technology Component (RTC) specification [9] from the Object Management Group (OMG), which defines a common introspection interface. Introspection is an important part of OpenRTM-aist, providing the basis upon which tools build to interact with and manage systems using the architecture.

The central concept in OpenRTM-aist is the RT Component, or RTC. Each RTC has an internal state machine with known states. The component's internal health can be both monitored and controlled through the introspection interface. In order for a component to begin executing, it must be activated, placing its state machine in the *Active* state. Execution can be terminated by deactivating the component, returning its state machine to the *Inactive* state. If an error occurs in the component, it moves to the *Error* state, from where it must be reset to move it back to the *Inactive* state.

OpenRTM-aist supports data-flow- and request-based communications, with interaction between components occurring at "ports." The component network is formed by making connections between the ports of the components. CORBA is used as the transport.

OpenRTM-aist can use manager daemons to load and manage components dynamically on remote nodes.

OpenRTM-aist uses CORBA [6] to implement the introspection interface. Components must register on a known name server (there can be more than one known name server in use). Typically, components are organised on the name server using naming contexts below the root context in order to provide hierarchical categorisation.

OpenRTM-aist is part of a larger project by Japan's New Energy and Industrial Technology Development Organisation (NEDO) for creating the elemental technologies of advanced robotics. Alongside OpenRTM-aist, there are also several tools for use with the architecture under development. These include RT-SystemEditor, the main tool for creating and interacting with RT Systems, the component networks created using OpenRTM-aist.

3 Interacting with component-based software systems

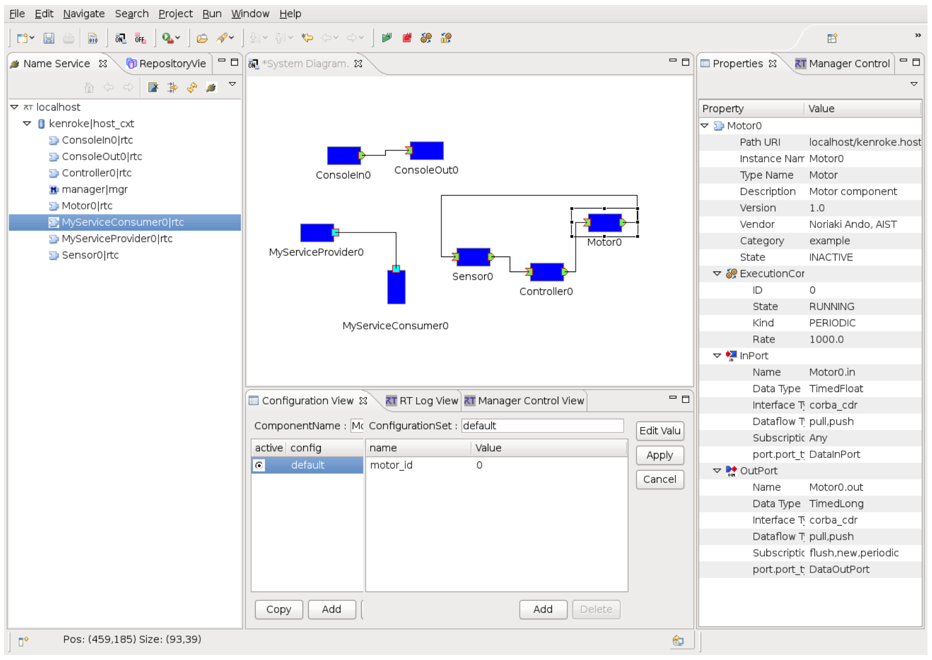
There are many approaches to constructing component-based systems from individual components. One is a fixed design of components with the connections between them hard-coded into the source code. Another is the use of configuration files specifying the network of connections between components. This method is used in, for example, ROS, where an XML-format file is used to describe the component network.

Graphical tools may be provided for working with the component-based software architecture. These tools are designed to simplify the development process to a drag-and-drop level, where new software systems can be constructed from existing software components by dragging them into a system diagram and drawing connections between them. An example of such a tool for a robot-oriented component architecture is RTSystemEditor, shown in Figure 1.

Graphical tools work well for system developers. With RTSystemEditor, developers are able to easily experiment with new system layouts through drag-and-drop interaction. A new system is created by dragging the desired components from the naming service view, which shows known components, and then drawing connections between their ports. An entirely new, complete system, featuring two components and a single connection, can be created in seconds. It is this speed benefit that has led to so much research in graphical tools over the years, and RTSystemEditor is not alone in providing this functionality. Other examples include EasyLab [5] and SmartSoft [10].

Unlike hard-coding and text-based configuration files, graphical tools may be useful beyond the creation step. RTSystemEditor, for example, can be used to manage a running RT System. Individual and groups of components can be started, stopped and reset, configuration parameters of components can be altered, links created and destroyed, and the health of each component monitored. This is possible because of the extensive introspection interface offered by OpenRTM-aist. The benefit to developers is the ability to modify the running RT System and experiment with different component network layouts in actual time. A developer can even use RTSystemEditor to monitor the state of the robot's software while it's running.

Unfortunately, the use of a graphical tool brings with it certain requirements that cannot always be met. GUIs introduce more difficulty than benefit in some situations commonly found in robotics:



Property	Value
Motor0	
Path URI	localhost/kenroke.host
Instance Nam	Motor0
Type Name	Motor
Description	Motor component
Version	1.0
Vendor	Noriaki Ando, AIST
Category	example
State	INACTIVE
ExecutionCor	
ID	0
State	RUNNING
Kind	PERIODIC
Rate	1000.0
InPort	
Name	Motor0.in
Data Type	TimedFloat
Interface T	corba_cdr
Dataflow T	pull,push
Subscriptic	Any
port.port_t	DataInPort
OutPort	
Name	Motor0.out
Data Type	TimedLong
Interface T	corba_cdr
Dataflow T	pull,push
Subscriptic	flush,new,periodic
port.port_t	DataOutPort

ComponentName : Mc ConfigurationSet : default

active	config	name	Value
●	default	motor_id	0

Pos: (459,185) Size: (93,39)

Fig. 1. The RTSystemEditor tool used with OpenRTM-aist to construct component networks.

– In low-resource environments, such as a robot's internal computer, the use of the graphical environment necessary to display the graphical tool, as well as the tool itself, can use up valuable resources. A robot's internal computer already has enough processing to do dealing with sensor data and planning. It often cannot handle the load of a graphical tool as well without compromising the capability of the robot.
– In order to get around the resource limitations, running the graphical tool from a remote computer may be possible. However, not all robot computers feature a network connection, making this method of connection impossible in those situations.
– Graphical tools cannot easily be scripted to perform repetitive tasks. Methods for scripting GUIs do exist, such as graphical tools with built in scripting systems and languages for manipulating GUIS [13]. However, these methods may not offer the flexibility that can be found in, for example, the command-line shell and tools available on the average UNIX system.

Inspired by the difficulty of using RTSystemEditor on our outdoor robots, where computing resources are tight and there is no network connectivity to a remote computer, as well as the massive flexibility of command-line tools on UNIX-based operating systems, we have experimented with alternative methods of creating and managing an OpenRTM-aist component network that do not rely on a graphical interface. We aim to create a method that allows for great flexibility in, as well as automation of, the management of the component network.

We must consider two aspects of the tool design. The first is how to reference the objects of the system. A graphical environment allows selection from a display; we must provide an approach that works in a text-based environment. The second is the method of control: pre-scripted with a programming language, or a more interactive approach, similar to what a graphical tool provides.

The following two sections describe the approach taken to each of these two issues.

3.1 The pseudo-file system

The RTSystemEditor tool presents known RTCs and other objects of the system in a tree structure. The top level of this tree is the root, and below that are known name servers, on which OpenRTM-aist objects register. Below the name servers are the objects themselves. They are typically sorted into naming contexts, which function as directories on the name servers.

The file system metaphor, commonly stated as "everything is a file," is a fundamental part of the UNIX philosophy. It says that everything on the system can be treated as a file. Inspired by this, we have used it to represent the same tree structure displayed in RTSystemEditor. The tools described in this paper present the user with a virtual file system. Files in the file system represent OpenRTM-aist objects such as RTCs and managers. Directories represent name servers and naming contexts. The user can address a component by providing its absolute path in the virtual file system, or a relative path from their current working directory.

3.2 Interacting with components

We could use a programming language to interact with the system. Creating a software library to interact with the components would support this. We would be able to program various interactions with the system objects. However, such a library already exists in the form of the architecture itself. It provides an API, defined in CORBA IDL, used to introspect the objects. Similar libraries exist for most middleware systems. The omniORB CORBA implementation [7], for example, has an extensive API that can be used to interact with CORBA objects at a low level. omniORB's API even already allows programmers to address objects registered on naming services using the path addressing scheme discussed in the previous section.

If such a library is already available, why doesn't the programming approach meet our needs? It doesn't because we desire the same level of interaction granted by graphical tools. Having to write a new program for everything we wish to do with the system is both inflexible and infeasible - developers do not have that much time.

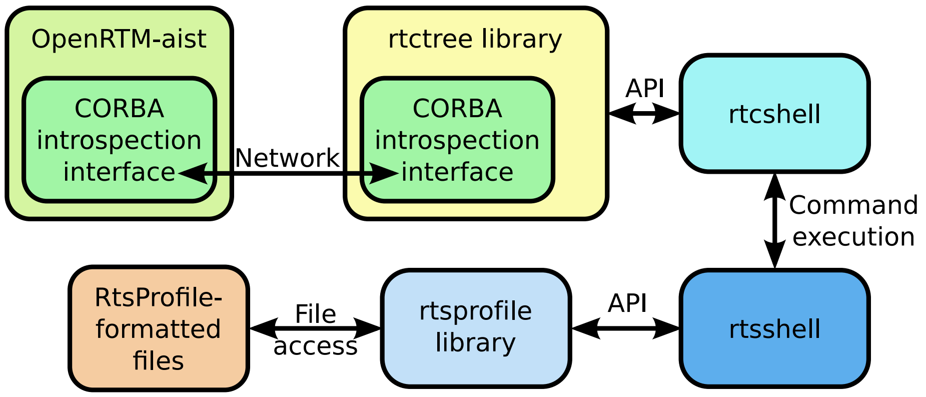


Fig. 2. The interaction between OpenRTM-aist and the various tools.

In order to achieve the flexibility desired, we need an interactive approach. Drawing further inspiration from UNIX, we have created two sets of command-line tools for manipulating the objects of an RT System. We have named these "rtcshell," for managing individual objects one at a time, and "rtsshell," for managing entire systems at once.

4 Shell utilities for OpenRTM-aist

The tools in both tool kits follow the UNIX philosophy of being small, doing one thing, doing it well, and using aggregation to add power. Each individual tool only performs one task, such as checking the life-cycle state of a component or making a connection, but they can be combined together and with existing UNIX commands to create more powerful tools.

The tool kits build on two libraries that were created to provide most of their functionality:

rtctree Implements the virtual file system and is responsible for all interaction with the OpenRTM-aist introspection interfaces.

rtsprofile Provides reading and writing of files in the RtsProfile format, an XML and YAML specification for describing RT Systems.

The interaction between the tool kits, the libraries and OpenRTM-aist is shown in Figure 2. All libraries and tools are implemented in Python.

4.1 rtctree

The rtctree library implements the virtual file system, which provides information about and addressing of all known OpenRTM-aist objects. The file system's

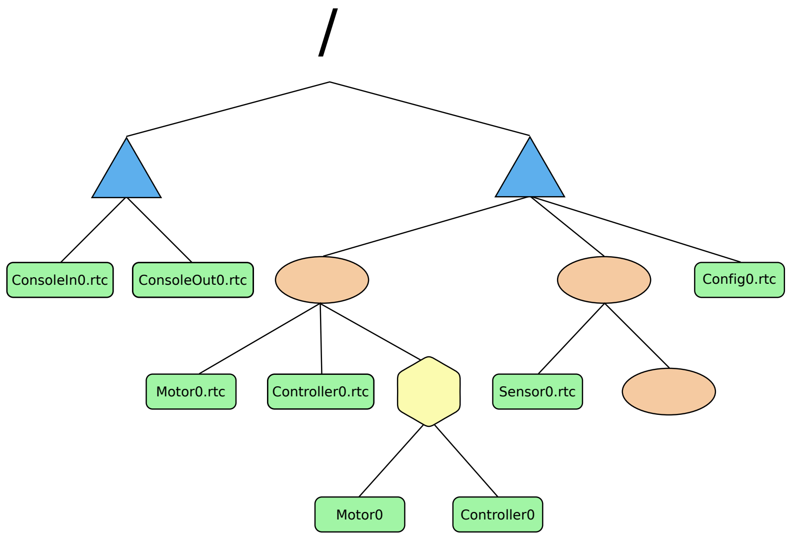


Fig. 3. The general structure of the pseudo-file system used by the *rtcshell* utilities, dubbed the *RTC Tree*. The root node is at the top of the file system tree. The blue triangles are naming service nodes. Red ovals are naming contexts below a root context. Green boxes are RT Components. The yellow hexagon is a manager. Below it are aliases to the two RT Components it is managing.

structure is illustrated in Figure 3. All paths branch off from a single root node, `/`. Below this root are all the known name servers. Name servers may have multiple naming contexts to which objects can register for organisational purposes. Within the file system, these are treated as directories.

The files of the virtual file system are RTCs and managers. The tools in rtcshell operate on these files. The files “contain” information about the object they represent, such as state and available ports for RTCs. This information is accessible using the various tools from rtcshell.

Manager objects are used to manage running component instances, so they contain references to the components they are managing. To make these easy to interact with, the virtual file system treats managers as directories containing aliases to the RTCs.

Using a tree structure allows additional tree-structure operations to be performed, such as iterating over the tree to perform an operation on every node matching certain criteria.

4.2 rtcshell

The rtcshell tool kit contains a set of shell commands for interacting with individual RTCs and managers. It uses the rtctree library to access the introspection interfaces of OpenRTM-aist objects. Each command in the tool kit has one specific function. These can be roughly divided into categories: navigating the virtual file system, viewing “file” contents, and manipulating files.

Navigating the virtual file system Navigation around the virtual file system is essential for making the tools usable. We cannot expect the user to supply a full path to every command.

The virtual file system is not a part of the real file system[1]. rtcshell therefore needs to provide its own commands for changing the virtual working directory. It provides equivalent to standard commands such as `cd` and `ls`. These are listed below.

rtcwd Changes the working directory to the given path, which may be absolute or relative to the current working directory.
rtpwd Prints the current working directory.
rtls Lists the contents of a given path or the current working directory.
rtfind Searches the virtual file system for objects and directories matching given search criteria. This command is implemented using the iteration function provided by rtctree.

The current working directory is stored in an environment variable, `RTCSH_CWD`. This variable is manipulated by `rtcwd` and used by the other commands of rtcshell when using paths.

Examples are shown in Listing 1.1. Lines 1 to 7 show using `rtls`. Like the standard `ls` command, `rtls` has both short and long forms. The long form briefly displays some useful information about components, such as their current state and the number of connections present. It is useful for monitoring the overall state of running components.

Listing 1.1. Examples of using the rtcshell tools to operate on the virtual system. Some lines have been removed for brevity.

```
1  $ ./rtls
2  Clusterer0.rtc     Hokuyo_AIST0.rtc
   kenroke.host_cxt/
4  $ ./rtls -l
   Inactive  2/0  1/0  1/0  0/0  Clusterer0.rtc
6  Inactive  4/0  0/0  3/0  1/0  Hokuyo_AIST0.rtc
   -         -    -    -    -    kenroke.host_cxt
8  $ rtcwd kenroke.host_cxt/
   $ rtcat ConsoleIn0.rtc
10 ConsoleIn0.rtc  Inactive
     Category       example
12   Description    Console input component
     Instance name  ConsoleIn0
14   Parent
     Type name      ConsoleIn
```

[1] We are investigating methods for making it a part of the real file system, similar to the /sys virtual file system in Linux.

```
16   Vendor          Noriaki Ando, AIST
     Version         1.0
18  +Execution Context 0
    +DataOutPort: out
20 $ rtconf ConfigSample0.rtc set default int_param0 5
21 $ rtconf ConfigSample0.rtc set int_param1 3
22 $ rtconf ConfigSample0.rtc list -l
   -default*
24   double_param0  0.11
     double_param1  9.9
26   int_param0     5
     int_param1     3
28   str_param0     foo
     str_param1     bar
30   vector_param0  0.0,1.0,2.0,3.0,4.0
31 $ rtact SequenceInComponent0.rtc
32 $ rtls -l
   Inactive  8/0  0/0  8/0  0/0  SequenceOutComponent0.rtc
34 Active    8/1  8/1  0/0  0/0  SequenceInComponent0.rtc
   [...]
36 $ rtcon ConsoleIn0.rtc:out ConsoleOut0.rtc:in
37 $ rtcat ConsoleIn0.rtc -l
38 ConsoleIn0.rtc  Inactive
   [...]
40  -DataOutPort: out
       dataport.data_type           TimedLong
42     dataport.dataflow_type       push
       dataport.interface_type      corba_cdr
44     dataport.subscription_type   flush,new,periodic
       port.port_type               DataOutPort
46    +Connected to
         /localhost/ConsoleOut0.rtc:in
```

Viewing “file” information The files of the virtual file system “contain” information from the objects they represent, such as the state of an RTC or the list of loaded modules for a manager. rtcshell provides several tools for viewing this information.

rtcat This is the most basic command for viewing contents. It prints out the information provided by an object’s introspection interface, showing information such as an RTC’s current state, its available ports, and the connections provided on those ports. An example is shown in Listing 1.1, lines 9 to 19.

rtconf Displays the configuration parameters of RTCs. Each RTC can contain various parameters used by its internal behaviour. See Listing 1.1, lines 20 to 30 for an example.

rtprint This command displays the data being sent over a port. It is useful for checking if an RTC is sending the information the user expects. It makes use of Python’s ability to convert nearly any object to a string.

Manipulating files The most important function of rtcshell is manipulating the objects of the virtual file system, particularly for connecting ports and changing the state of RTCs. The following commands are used for this.

rtact Activates a component, causing it to begin executing. See Listing 1.1, lines 31 to 35 for an example.
rtdeact Deactivates a component, halting its execution.
rtreset Resets a component, moving it back to the *Inactive* state after an error has occurred.
rtcon Connects two ports together. Ports of a component are specified after a colon at the end of a component's path. See Listing 1.1, lines 36 to 47.
rtdis Removes connections.
rtinject Injects data into a port. This is useful to test a component's response to inputs without needing to implement another component to create test data.
rtmgr Manipulates manager objects. Managers are daemons used to deploy components. `rtmgr` can be used to control the deployment of RTCs loaded from shared libraries.

4.3 rtsshell

rtcshell is only used for manipulating individual file system objects. It does not provide any facilities for easily manipulating entire RT Systems with a single command. rtsshell is a complimentary tool kit that provides this functionality. It works with RtsProfile files using the rtsprofile library to describe complete systems, and uses rtcshell to manipulate the individual RTCs and managers.

rtcryo Creates a new RtsProfile file from the currently-running RT System.
rtteardown Removes all connections in an RT System.
rtresurrect Uses an RtsProfile file to recreate an RT System from running RTCs by restoring connections and configuration parameters.
rtstart Starts an RT System by shifting all RTCs to the *Active* state.
rtstop Stops an RT System by shifting all RTCs to the *Inactive* state.

All these commands are capable of checking if the necessary objects are available in the virtual file system. They can also use a partially-ordered planner to ensure that state change commands are executed in the correct order, as specified by the developer when creating the RT System.

5 Discussion

Each tool performs one task only. However, while each tool is simplistic by itself, the collection as a whole is both flexible and powerful. As is common for UNIX tools, aggregation of the tools provides additional power.

For example, combining the standard UNIX `watch` command and `rtls -l` gives a continuously-updating display of component state. A list of all output ports of a component can be obtained by combining `rtcat` and `grep`. The shell script `for` command can be combined with `rtfind` and `rtact` to activate all components matching a given name specification. The implementation of the rtsshell tools relies on aggregating the rtcshell tools to perform their actions, such as starting all components in the system using `rtact`. We do note that this usage style leads to the RTC Tree being constructed over and over, placing additional load on the introspection interfaces of components. This can be mitigated by careful management of the rate at which tools are executed.

These tools are different from RTSystemEditor. Many of the capabilities are the same, but each has its own advantages. rtcshell is not useful for visualising large component networks, something RTSystemEditor's graphical interface excels at. RTSystemEditor is very difficult to automate, while it is trivial to automate the command-line tools using shell scripting.

There is little directly-comparable work in robotics. Some other architectures have tools with some related capabilities. For example, the Task Browser component from Orocos [3] allows navigation around a running network of components, viewing the ports, properties and so on. This is more like an interactive component for debugging other components than a tool for creating and managing component-based systems.

ROS includes introspection of its communications channels, and provides shell tools for viewing information about them. One such tool is `rostopic`, which can show various information about a topic such as the data being transmitted over it. However, these tools can only monitor and send data to topics. They cannot manipulate the component network, such as creating new connections, nor can they find any information about components themselves.

YARP[4] contains command line tools for tasks such as connecting ports together and querying port locations (similar to the `rtcon` tool), through a central server.

The strength of rtcshell and rtsshell is in quickly creating small systems for experimentation, for managing both large and small RT Systems, and for automation of common tasks. No other system currently matches all of their functionality.

6 Conclusions

This paper has described a set of command-line tools for managing RT Components and component networks for the OpenRTM-aist architecture. The tools treat known components and other OpenRTM-aist objects as part of a file system. They allow the user to easily inspect and manage components in a console.

This form of interaction is well suited to the low-resource environments that are commonly found in robotics, where a resource-intensive graphical tool is not feasible. They allow greater freedom for developers. The tools can be scripted using standard shell scripting facilities, facilitating the automation of tasks.

We believe that such a set of command-line tools adds additional usability to component-based software architectures. The UNIX philosophy of tools being small with power through aggregation has been proved over the years to lead to a highly-flexible system. Its application to robotics is important if robots are to be likewise flexible, maintainable and easy to develop.

Acknowledgements

This research was partially supported by the Grant-in-Aid for the Developing Intelligent Technology for Next-Generation Robots Project (FY2007-2011) of the New Energy and Industrial Technology Development Organization of Japan.

References

1. Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., Yoon, W.K.: RT-middleware: distributed component middleware for RT (robot technology). In: Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on. pp. 3933–3938 (August 2005)
2. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Oreback, A.: Towards component-based robotics. In: Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on. pp. 163–168 (Aug 2005)
3. Bruyninckx, H.: Open robot control software: the orocos project. In: Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on. vol. 3, pp. 2523 – 2528 vol.3 (2001)
4. Fitzpatrick, P., Metta, G., Natale, L.: Towards long-lived robot genes. Robotics and Autonomous Systems 56(1), 29 – 45 (2008)
5. Geisinger, M., Barner, S., Wojtczyk, M., , Knoll, A.: A software architecture for model-based programming of robot systems. In: Kröger, T., Wahl, F.M. (eds.) Advances in Robotics Research - Theory, Implementation, Application, pp. 135–146. Springer-Verlag Berlin Heidelberg, Braunschweig, Germany (June 2009)
6. Henning, M., Vinoski, S.: Advanced CORBA Programming with C++. Addison-Wesley Professional (1999)
7. omniORB. `http://omniorb.sourceforge.net/` (2010)
8. ROS Wiki. `http://www.ros.org` (2010)
9. The Robotic Technology Component Specification - Final Adopted Specification. `http://www.omg.org/technology/documents/spec_catalog.htm` (2010)
10. Schlegel, C., Hassler, T., Lotz, A., Steck, A.: Robotic software systems: From code-driven to model-driven designs. In: Advanced Robotics, 2009. ICAR 2009. International Conference on. pp. 1–8 (June 2009)
11. Song, B., Jung, S., Jang, C., Kim, S.: An Introduction to Robot Component Model for OPRoS (Open Platform for Robotic Services). In: Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots 2008, Workshop Proceedings of. pp. 592–603 (Nov 2008)
12. Szyperski, C., Gruntz, D., Murer, S.: Component Software – Beyond Object-Oriented Programming. Addison-Wesley and ACM Press, 2nd edn. (2002)
13. Visual Basic Developer Center: `http://msdn.microsoft.com/vbasic/` (2010)