Best Practice in Robotics

Grant Agreement Number: 231940

Funding Period: 01.03.2009 – 28.02.2013

Instrument: Collaborative Project (IP)

Deliverable D-2.1

Best Practice Assessment of Software Technologies for Robotics

Author names:	Azamat Shakhimardanov Jan Paulus Nico Hochgeschwender Michael Reckhaus Gerhard K. Kraetzschmar
Lead contractor for this deliverable:	Bonn-Rhein-Sieg University (BRSU)

Due date of deliverable:	01.03.2010
Actual submission date:	01.03.2010
Dissemination level:	PU/RE
Revision:	1.0

Abstract

The development of a complex service robot application is a very difficult, time-consuming, and error-prone exercise. The need to interface to highly heterogeneous hardware, to run the final distributed software application on a ensemble of often heterogeneous computational devices, and to integrate a large variety of different computational approaches for solving particular functional aspects of the application are major contributing factors to the overall complexity of the task. While robotics researchers have focused on developing new methods and techniques for solving many difficult functional problems, the software industry outside of robotics has developed a tremendous arsenal of new software technologies to cope with the ever-increasing requirements of state-of-the-art and innovative software applications. Quite a few of these techniques have the potential to solve problems in robotics software development, but uptake in the robotics community has often been slow or the new technologies were almost neglected altogether.

This report identifies, reviews, and assesses software technologies relevant to robotics. For the current document, the assessment is scientifically sound, but neither claimed to be complete nor claimed to be a full-fledged quantitative and empirical evaluation and comparison. This first version of the report focuses on an assessment of technologies that are generally believed to be useful for robotics, and the purpose of the assessment is to avoid errors in early design and technology decisions for BRICS.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant Agreement No. 231940.

Contents

Executive Summary 1

1 Introduction 3

1.1 The BRICS Project Context . . . 3
1.2 Motivation for Assessing Software Technologies . . . 4
1.3 Objectives of the Assessment Exercise . . . 5
1.4 Overview on the Report . . . 5

2 Component Technologies 7

2.1 Component Models in Robotics Software . . . 8
2.1.1 OROCOS . . . 8
2.1.2 OpenRTM . . . 12
2.1.3 Player . . . 16
2.1.4 ROS . . . 20
2.2 Analysis and Comparison of Component Models . . . 23
2.2.1 Introduction and Essential BCM . . . 23
2.2.2 ORCA2 . . . 25
2.2.3 OpenRTM . . . 26
2.2.4 GenoM . . . 27
2.2.5 OPRoS . . . 28
2.2.6 ROS . . . 29
2.2.7 OROCOS . . . 30
2.2.8 Summary of Comparative Analysis . . . 30
2.3 Conclusions . . . 30

3 Communication Middleware 33

3.1 The Marketplace of Communication Middleware . . . 33
3.2 Assessment of Communication Middlewares . . . 34
3.2.1 Specification versus Implementation . . . 34
3.2.2 Data Distribution Service (DDS) . . . 37
3.2.3 Advanced Message Queuing Protocol (AMQP) . . . 37
3.2.4 Java Message Service (JMS) . . . 38
3.2.5 Message Passing Interface (MPI) . . . 38
3.2.6 Common Object Request Broker Architecture (CORBA) . . . 39
3.2.7 XML Remote Procedure Call (XML-RPC) . . . 39
3.2.8 Internet Communication Engine (ICE) . . . 39
3.2.9 Transparent Inter-Process Communication (TIPC) . . . 40
3.2.10 D-Bus . . . 40
3.2.11 Lightweight Communications and Marshalling (LCM) . . . 41
3.2.12 Spread Toolkit (Spread) . . . 41

3.3 Conclusions . . . 41

4 Interface Technologies **43**
4.1 Component Interface Models . . . 43
4.2 Hardware Device Interfaces and Abstractions . . . 49
4.2.1 The Need for Hardware Abstractions . . . 49
4.2.2 Challenges for Hardware Abstraction . . . 49
4.3 Assessment of Hardware Abstraction Approaches . . . 51
4.3.1 Orca . . . 51
4.3.2 ROS . . . 52
4.3.3 Player . . . 53
4.3.4 YARP . . . 54
4.3.5 ARIA . . . 55
4.3.6 Mobile Robot Programming Toolkit (MRPT) . . . 55
4.3.7 OPRoS . . . 56
4.3.8 OpenRTM . . . 56
4.4 Conclusions . . . 59

5 Simulation and Emulation Technologies **60**
5.1 Introduction . . . 60
5.2 Simulators Used in Robotics . . . 62
5.2.1 Low level Simulators . . . 62
5.2.2 Algorithm Evaluation . . . 63
5.2.3 Robot System Evaluation . . . 63
5.2.4 Learning Algorithms . . . 64
5.2.5 Learning Team Behaviors . . . 65
5.3 Conclusions . . . 65

6 Conclusions **67**
6.1 Component Technologies . . . 67
6.2 Communication Middleware . . . 68
6.3 Interface Technologies . . . 68
6.4 Simulation and Emulation Technologies: . . . 69

References **70**

List of Figures

2.1 Interface types in the OROCOS component model. 9
2.2 OROCOS component state machine models, as defined by class `TaskContext` [1, 2]. 10
2.3 OROCOS component implementation model. 10
2.4 The OpenRTM component implementation model. 13
2.5 The OpenRTM component state machine model[3]. 14
2.6 The Player server implementation model. 16
2.7 Player component model. 19
2.8 ROS pseudo-component model. 21
2.9 Simplified brics component model. 24

4.1 Meta-component model. 45
4.2 Interface separation schema. 45
4.3 Topology representing functional interfaces of components. 46
4.4 Interface categorization according to data and execution flows 46
4.5 Component interface separation scheme according to 4C concerns. 47
4.6 Orca range finder hardware hierarchy 52
4.7 YARP Hardware Abstraction Hierarchy 54
4.8 ARIA hardware hierarchy for range devices 55
4.9 MRPT hardware hierarchy 55
4.10 OPRoS hardware hierarchy 56

List of Tables

2.1 Mapping between BRICS CMM constructs and other systems. 25
2.2 Mapping between BCM constructs and ORCA2. 26
2.3 Mapping between BRICS component model constructs and OpenRTM. 27
2.4 Mapping between BRICS CMM constructs and GenoM. 28
2.5 Mapping between BRICS CMM constructs and OPRoS. 29
2.6 Mapping between BRICS CMM constructs and ROS. 30
2.7 Mapping between BRICS CMM constructs and OROCOS. 30
2.8 Comparison of component modelling primitives in different robot software systems with respect to the BRICS component model. 31

3.1 Compiled list of different middleware technologies 35
3.2 Middleware specifications and their associated implementations. 35
3.3 Middleware implementations and their supported programming languages. 36
3.4 Middleware implementations and their supported platforms. 36

4.1 OpenRTM range sensing device interfaces 57
4.2 Hardware Abstraction Comparison Table 58

Executive Summary

This study identifies four major technologies that can improve the software development process in robotics:

- Component-based programming technology has the potential to significantly increase portability, interoperability, and reuse of software modules in robotics. There is currently no component model perfectly well suited for robotics, but some candidates are not so far away. BRICS should make an attempt to progress this work by defining a component model taking into account the requirements identified during this study.
- Communication middleware is a necessary element in any state-of-the-art robot control architecture, because robotics control software for non-trivial robots requires distributed computing. None of the established middlewares is currently predominant, and it is unpredictable, whether any of the currently available systems will become a de facto standard. BRICS should follow an open approach with respect to middleware and design software such that middleware systems can be replaced with reasonable effort.
- Interface technologies play a central role in taming the high negative impact of hardware heterogeneity. For every hardware component available and used in robotic platforms, their often peculiar custom-defined interface should be cleanly encapsulated by object-oriented classes. Abstraction hierarchies built upon these classes allow the definition of more abstract, generic interfaces. Programming against abstract interfaces can significantly enhance the generality and reusability of functional modules in robotic control. Some robotics software frameworks already provide a few such abstractions, but there is neither a thorough approach to design clean object-oriented interfaces nor a systematic attempt to build interface abstraction hierarchies. BRICS should do exactly this, and do it systematically.
- Simulation and emulation technologies should be used to enable simultaneous development of robotic hardware and software and to improve the quality and safety of the resulting code. Although many simulators are available, they target many different things. Due to necessary tradeoffs between model precision and simulation performance, it is unlikely that a single simulator meeting all requirements in a suitable way can be designed. Thus, BRICS should consider the use of different simulators/emulators at different stages of software development. A primary objective to have in mind is that robot control software should be seamlessly usable on both the actual hardware platform and in the simulator.

Chapter 1

Introduction

The work described in this report was performed in the context of the EU project BRICS. We briefly describe this project context, then motivate why an assessment of software technology is appropriate and what the objectives of this assessment are. Finally, we survey the structure of this report.

1.1 The BRICS Project Context

BRICS[1] addresses the urgent need of the robotics research community for common robotics research platforms, which support integration, evaluation, comparison and benchmarking of research results and the promotion of best practice in robotics. In a poll in the robotics research community performed in December 2007 95% of the participants have called for such platforms. Common research platforms will be beneficial for the robotics community, both academic and industrial. The academic community can save a significant amount of resources, which typically would have to be invested in *from scratch developments* and *me-too approaches*.

Furthermore, scientific results will become more comparable which might promote a culture of sound experimentation and comparative evaluation. Jointly specified platforms will foster rapid technology transfer to industrial prototypes, which supports the development of new robotics systems and applications. This reduces the time to market and thereby gives the industrial community a competitive advantage. To achieve these objectives the BRICS project proposes the development of a design methodology, which focuses on three fundamental research and development issues. This methodology will be implemented in three interwoven lines of technical activities:

- Identification of best practice in robotics hardware and software components
- Development of a tool chain that supports rapid and flexible configuration of new robot platforms and the development of sophisticated robot applications
- Cross-sectional activities addressing robust autonomy, openness and flexibility, and standardisation and benchmarking

The authors of this report all work at Bonn-Rhein-Sieg University of Applied Sciences (BRSU), which is the partner in BRICS responsible for *Architecture, Middleware, and Interfaces.* This work package is to provide fundamental software components using state-of-the-art software technologies and the usage of these components needs to be well embedded into the tool chain.

The BRICS project is of fundamental importance to ensure the sustained success and competitiveness of European robotics research and the European robotics industry.

[1]This section is a modest revision of the *BRICS in a nutshell* section of the project proposal.

1.2 Motivation for Assessing Software Technologies

Software development for robotics applications is a very time-consuming and error-prone process. Previous work described in the literature[4] identifies three major sources responsible for the complexity of software development in robotics:

Hardware Heterogeneity: A reasonably complex service robot integrates an arsenal of sensors, actuators, communication devices, and computational units (controller boards, embedded computers) that covers a much wider variety than most application domains. While e.g. aviation and the automotive industry face similar situations, their development processes are much more resourceful and use a significant number of specialists to manage the problems arising from heterogeneity. Robotics, esp. service robotics and robot applications targeting a consumer market are far from enjoying a similarly comfortable situation.

Distributed Realtime Computing: Non-trivial service robots almost inevitably use several networked computational devices, all of which run some part of the overall control architecture, which must work together in a well-coordinated fashion. Applying concepts and principles of distributed computing is therefore a must. That the networking involves several kind of communication devices and associated protocols adds even more complexity. It is not unusual at all that a service robot forces a developer to handle three to five different communication protocols. Furthermore, some devices may pose rather strict timing constraints, and the developer must deal with realtime computing issues on top of all other complexities.

Software Heterogeneity: A full-fledged service robot must integrate a variety of functionalities that include basically every sub-area of Artificial Intelligence, including knowledge representation and reasoning, probabilistic reasoning, fuzzy logic, planning and scheduling, learning and adaptation, evolutionary and genetic computation, computer vision, speech processing and production, natural language understanding and dialogue, sensor interpretation and sensor fusion, and intelligent control. Many of these areas have developed their own set of computational methods for solving their respective problems, often using very different programming languages, data structures and algorithms, and control approaches. Integrating the different functionalities into a coherent and well-coordinated robot control architecture is a substantial challenge.

Since about a decade there is a rising interest in the robotics community to improve the robot application development process. Aside of improving the development process itself (an accompanying report *Best Practice Assessment of Software Engineering Methods*, [5] will be forthcoming) and providing a suitable tool chain to support the process (see work performed in WP 4), the use of state-of-the-art and innovative software technology is of prime interest to improve the effectivity and efficiency of the development process and the quality of the resulting product.

The main question then is: Which software technologies can help the robotics community to overcome the problems arising from the complexity of the robot application development process?

There are three main aspects to cover when answering this question. The first aspect is identifying software technologies that have the potential to address one or more of the problems faced by robot application developers. The second aspect is assessing to what extent current robotics software development frameworks already make use of these technologies. And the third aspect is to look at standardization activities, which are underway even in some robotics communities. Neglecting existing or ongoing standardization efforts could prevent project results from becoming adapted by a wider audience. Identifying and criticizing serious deficiencies in standard proposals under discussion could help to improve standards and make them more effective.

Note, that this report does not cover issues related to robot control architectures; these aspects will be considered and discussed in a future report.

1.3 Objectives of the Assessment Exercise

This report has several objectives:

- Identify well-known as well as new, innovative software technologies that are relevant for robotics.
- Survey the conceptual and technical approach of such software technologies.
- Discuss their innovation potential and how they can benefit the robot application development process.
- Assess whether and to what extent they have already been taken up inside robotics or outside, e.g. in closely related fields like automotive engineering, aerospace engineering, or advanced applications of embedded systems, such as sensor networks, and summarize the experiences made so far.
- Identify problems or deficiencies that may present obstacles for take-up and adoption of the technology.
- Derive a first action plan for how to proceed with the technology in BRICS.

1.4 Overview on the Report

The remainder of the report is structured as follows: The next four chapters describe four classes of software technologies that have been identified and are expected to be of major importance for BRICS:

- **component-based software technologies**,
- **communication middleware**,
- **interface technologies**, and
- **simulation and emulation technologies**.

The final chapter draws some conclusions and describes the implications for BRICS.

Chapter 2

Component Technologies

In recent years, advancements in hardware, sensors, actuators, computing units, and — to a lesser extent — in software technologies has led to the proliferation of robotics technology. More complex and advanced hardware required increasingly sophisticated software infrastructures in order to operate properly. This software infrastructure should allow developers to tackle the complexity imposed by hardware and software heterogeneity and distributed computing environments. That is, the infrastructure software should provide a structured organization of system elements/primitives in the form of independent software units and support communication among these independent software units.

Robotics is not the only field which needs such infrastructure. Others include telecommunication, aerospace, and automotive engineering. An analysis of the approaches used in these domains shows that the two essential ingredients for developing distributed software systems are *i)* the definition of a structured software unit model, often referred to as *component model*, and *ii)* the the definition of a communication model for modeling communication between these units. A software unit model includes strictly defined interface semantics and a method for describing the internal dynamical structure of a unit, whereas a communication model, though often not as clearly defined as the software unit, requires the specification of inter-unit interaction styles and specific transport protocol types.

Before categorizing and analyzing existing component models, we distinguish component-oriented programming (COP) from object-oriented programming (OOP). Basically, all the objectives of OOP are also an objective in COP, but COP development was driven by the desire to fix some loose ends of OOP. The following attributes of COP are emphasized in a comparison with OOP in [6]:

- COP emphasizes encapsulation more strongly than OOP. COP differentiates more strictly between interface design and implementation of the specified functionality than OOP. The user of a component does not see any details of the component implementation except for its interfaces, and the component could be considered as a piece of pre-packaged code with well-defined public access methods. This separation can also be followed in OOP, but is not as directly enforced, e.g. by the programming language.

- COP focuses more on packaging and distributing a particular piece of technology, while OOP is more concerned about how to implement that technology.

- COP components are designed to follow the rules of the underlying component framework, whereas OOP objects are designed to obey OO principles.

One can also differentiate three factors exposing advantages of COP over OOP with respect to the composition of units adopted in each approach:

- Reusability levels
- Granularity levels
- Coupling levels

These factors are tightly related to each other [6].

The fundamental concept of component-oriented programming is the component and their interfaces. Approaches can differ quite widely regarding their ability to hierarchically compose more complex components from existing components.

Equally important as the component concept itself is the supportive infrastructure that comes with it, i.e. the framework for component instantiation, resource management, inter-component communication and deployment. These frameworks often are already equipped with the means for inter-component communication, which can influence interface design decisions to some extent. Unfortunately, it is often quite difficult or impossible to clearly delineate and decouple the component interaction model from the communication model. Each framework usually features its own component model, communication model, and deployment model. Different frameworks have different models for each of the above constituents. As a consequence, it is often not possible to interoperate and integrate them. In our analysis we mostly emphasize two of these models, the component model and the communication model.

The next section provides an almost exhaustive survey on existing component models used in robotics software. We analyze what the specific component features are and how these frameworks are implemented wrt. these models. Additionally, we provide a comparative analysis of the component interface categorization schemes in Section 4.1. Since different interface semantics define how a component interacts with other components and produces/consumes data, the type of communication policies and the infrastructure are the other elements looked into.

2.1 Component Models in Robotics Software Frameworks

2.1.1 OROCOS

The main goal of the OROCOS project is to develop a general purpose modular framework for robot and machine control. The framework provides basically three main libraries [7, 1, 2]:

1. The *Real-Time Toolkit (RTT)* provides infrastructure and functionality to build component based real-time application.
2. The *Kinematics and Dynamics Library (KDL)* provides primitives to calculate kinematic chains.
3. The *Bayesian Filtering Library (BFL)* provides a framework to perform inference using Dynamic Bayesian Networks.

The RTT provides the core primitives defining the OROCOS component model and the necessary code base for the implementation of components. Figure 2.1 illustrates that the OROCOS component model supports five different types of interfaces. This interface separation fits well into data and execution flow, as well as time property-oriented schema (see Section 4.1). In the former case, commands, methods, and events make up execution flow, whereas data ports represent data flow of the component. If one looks at temporal properties of the interfaces, synchronous (method and data) and asynchronous interfaces (command and event) can be distinguished.

- *Commands* are sent by a component to other components (receivers) to instruct them to achieve a particular goal. They are asynchronous, i.e. caller does not wait till it returns.

- *Methods* are called by other components on a particular component to perform some calculations. They are synchronous.
- *Properties* are runtime modifiable parameters of a component which are stored in XML format.
- *Events* are handled by a component when a particular change occurs (event/signal is received).
- *Data Ports* are thread-safe data transport mechanism to communicate (un)buffered data among components. Data port-based exchange is asynchronous/non-blocking.

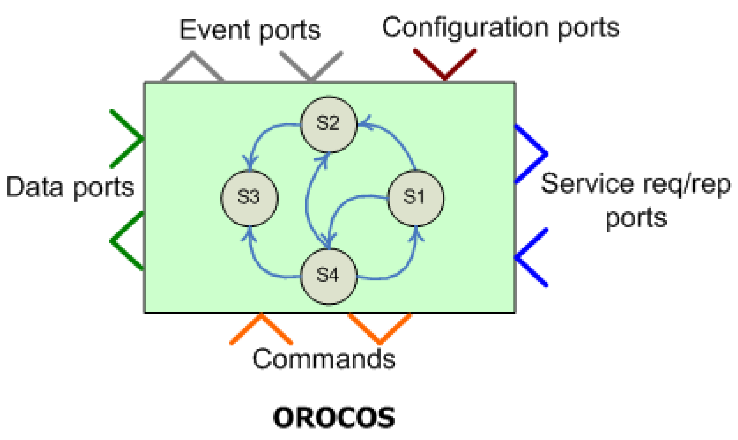
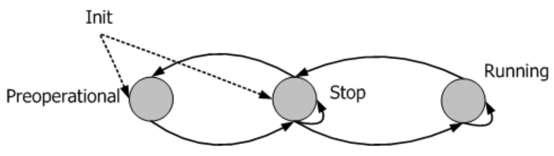


Figure 2.1: Interface types in the OROCOS component model.

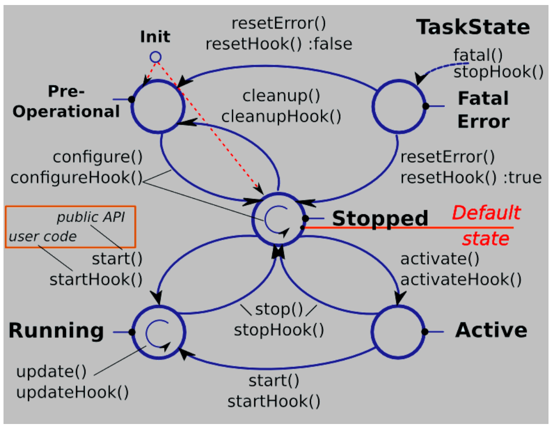
The application programmer point of view: In OROCOS, a component is based on the class `TaskContext` 2.3. The component is passive until it is deployed (an instance of the class is created), when it is allocated in its own thread. The `TaskContext` class defines the "context" in which the component task is executed. A task is a basic unit of functionality which is executed as one or more programs (a C function, a script, or a state machine) in a single thread. All the operations performed within this context are free of locks, priority inversions, i.e. thread-safe deterministic. This class, from the implementation point of view, serves as a wrapper class (itself composed of several classes) for other sub-primitives of a component, such as ports, thread contexts, operation processors, that is, it presents the composition of separate functionalities/interfaces 2.3 .

One of the main classes in `TaskContext` is the `ExecutionEngine`. It executes the decision logic defined by `TaskContext` by executing programs and state machines 2.2 . The type of execution can be defined through `ActivityInterface`. It allows execution in *Sequential, Periodic, Nonperiodic* and *Slave* modes. Additionally, `ActivityInterface` allocates a thread to the `ExecutionEngine`, by default execution proceeds in a *Sequential* mode [1, 2].

In addition to `ExecutionEngine` that determines a coordination aspect of an OROCOS component, `TaskContext` implements `OperationInterface`, which is a placeholder for commands, methods, events, and attributes of the OROCOS component. Each of these operations is in turn processed by its own processor, e.g. for events it is `EventProcessor` and for commands it is `CommandProcessor`. From the implementation point of view these processors implement the same interface as the `ExecutionEngine` of the component. Therefore, it can be concluded that OROCOS has at least a two-level coordination through state machines: (a) on the level



(a) simplified



(b) expanded

Figure 2.2: OROCOS component state machine models, as defined by class `TaskContext` [1, 2].

of `TaskContext`, which is also main execution model of the component, and (b) on the level of operation processors [1, 2].

From the structural and deployment point of view the OROCOS component is a composite class which has its own thread of execution and could be deployed as a shared object or an executable. Listing 1 provides the signature of the TaskCore class, which serves as a basis for the internal OROCOS component model. This is achieved through the statically defined state machine structure. Listing 2 shows an example of a OROCOS component-based application.

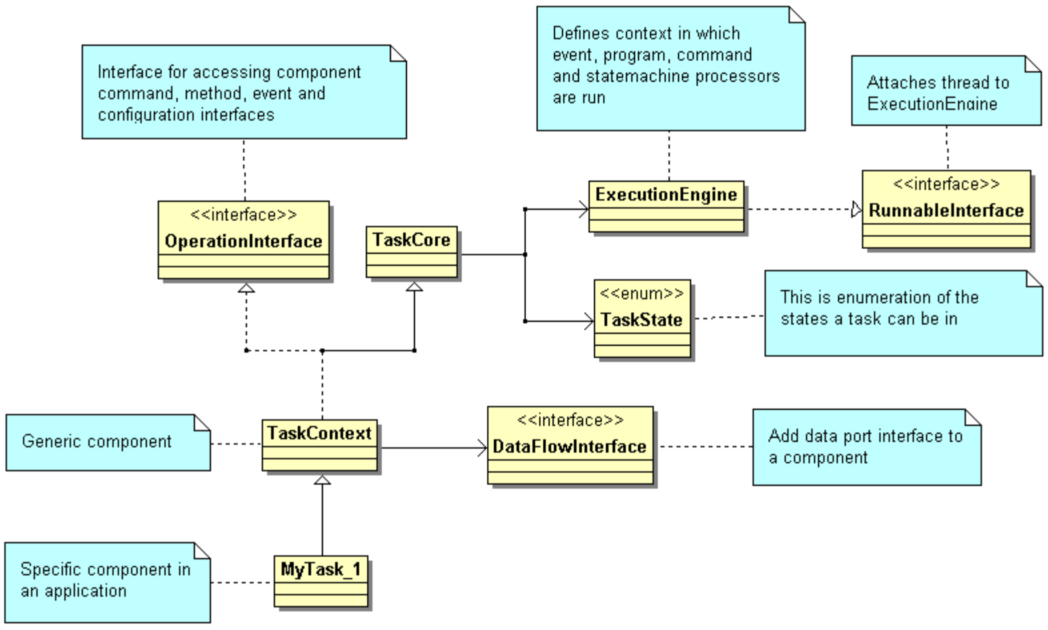

Interface for accessing component command, method, event and configuration interfaces
Defines context in which event, program, command and statemachine processors are run
Attaches thread to ExecutionEngine
<<interface>>
OperationInterface
TaskCore
ExecutionEngine
<<interface>>
RunnableInterface
<<enum>>
TaskState
This is enumeration of the states a task can be in
Generic component
TaskContext
<<interface>>
DataFlowInterface
Add data port interface to a component
Specific component in an application
MyTask_1

Figure 2.3: OROCOS component implementation model.

```
class RTT_API TaskCore
{
  public:
  enum TaskState
  { Init,
    PreOperational,
    FatalError
    Stopped,
    Active
    Running,
    RunTimeWarning,
    RunTimeError
  };

  virtual bool configure();
  virtual bool activate();
  virtual bool start();
  virtual bool stop();
  virtual bool cleanup();
  virtual bool resetError();
  virtual bool isConfigured() const;
  virtual bool isActive() const;
  virtual bool isRunning() const;

  virtual bool inFatalError() const;
  virtual bool inRunTimeWarning() const;
  virtual bool inRunTimeError() const;
  virtual bool doUpdate();
  virtual bool doTrigger();
  ...

  protected:
  virtual bool configureHook();
  virtual void cleanupHook();
  virtual bool activateHook();
  virtual bool startHook();
  virtual bool startup();
  virtual void updateHook();
  virtual void errorHook();
  virtual void update();
  virtual void stopHook();
  virtual void shutdown();
  virtual bool resetHook();
  virtual void warning();
  virtual void error();
  virtual void fatal();
  virtual void recovered();
};
```

Listing 1. OROCOS task signature

```
int ORO_main(int arc, char* argv[])
{
  MyTask_1 a_task("ATask");
  MyTask_2 b_task("BTask");

  a_task.connectPeers( &b_task );
  a_task.connectPorts( &b_task );

  a_task.setActivity( new PeriodicActivity(OS::HighestPriority, 0.01 ) );
  b_task.setActivity( new PeriodicActivity(OS::HighestPriority, 0.1 ) );

  a_task.start();
  b_task.start();

  a_task.stop();
  b_task.stop();

  return 0;
}
```

Listing 2. OROCOS task-based application

2.1.2 OpenRTM

OpenRTM[] is open source implementation of the RT-Middleware specification and is developed by AIST, Japan. This specification defines the following three sub-specifications which are complementary to each other: (i) functional entities, i.e. components which perform system-related computations, (ii) communication entities, i.e. middleware which provides communication means for the components, and (iii) tool entities, i.e. a set of tools supporting system- and component-level design and runtime utilities. Some more detailed information about each of these specifications include:

- The *RT-Component Framework* provides a set of classes which can be used to develop stand-alone software components. Conceptually, any RT middleware component can be decomposed into two main parts: (a) The component core logic, which defines the main functionality of the component. All algorithms are defined within the core logic of the component. (b) The component skin, or wrapper, provides necessary means to expose the functionality provided by the core logic through the interfaces to the external world. The RT component framework provides a set of classes which enable this wrapping [3, 8].

- The *RT-Middleware*: By definition, an RT-component is a decoupled entity whose final form can be either a shared library or an executable type. In the Robot Technology package, an RT-component is defined as the aggregation of several classes without main function, so there is no explicit thread of execution attached to it (it is a passive functional unit without any thread of execution). That is where RT-middleware comes into play. RT-middleware makes a call to a component and attaches an execution context and an appropriate thread to it. By doing this, RT-middleware also takes the responsibility to manage that component's life cycle, i.e. to execute actions defined by a component and communicate with other peers. Note that there can be many instances of RT-component running. RT-Middleware takes care of the whole instantiation process. That is, the calls are made from the middleware to a component. OpenRTM implementation uses a CORBA-compliant middleware environment, omniORB [3, 9, 10, 11].

- *Basic Tools Group*: The Robot Technology software system also comes with a set of utilities which simplify development, deployment, and configuration of the applications developed. In case of the OpenRTM implementation, these tools are RtcTemplate, a component skeleton code generation tool, and RtcLink, a component-based software composition tool.

As it was indicated, an RT-component is defined through its computational part, which is a core logic, and a component skin, which exposes the functionality of the core logic. From a more detailed perspective these two constituents can be further decomposed into the following subcomponents [3, 8, 9, 10, 11].

- The *Component Profile* contains meta-information about a component. It includes such attributes as the name of the component, and the profile of the component's ports.

- The *Activity* is where main computation takes place. It forms a core logic of a component. Activity has a state machine and the core logic is executed as component which transits between states or is in a particular state of processing.

- The *Execution Context* is defined by a thread attached to a component. It executes defined activities according to the given state.

- *Component Interfaces:* OpenRTM components provide three types of interfaces to communicate with other components or applications (see Figure 2.4):
  - *Data Ports*, which either send or receive data in pull or push mode.
  - *Service Ports*, which are equivalent to synchronous method calls or RPCs.
  - *Configuration Interfaces*, in addition to data exchange and command invocation interfaces. The OpenRTM component model provides an interface to refer to and modify parameters of the core logic from outside the component. These parameters may include name and value, and an identifier name. Reconfiguration of these parameters can be performed dynamically at runtime.

Therefore, depending on the type of port used, a component can interact either through a client/server pattern for service ports, or through a publisher/subscriber pattern for data ports. It can be observed that the OpenRTM interface categorization, as in case of the OROCOS component, also fits to data and execution flow-based schemes as well as a concern-based scheme (Figure 4.2).

It is interesting to compare component models of OpenRTM and OROCOS projects (figures 2.3 and 2.4). The OROCOS component model provides a more fine-grained set of interface types. That is, there are specific interfaces for particular types of operations. This allows a better control over interactions of a component with other peers.

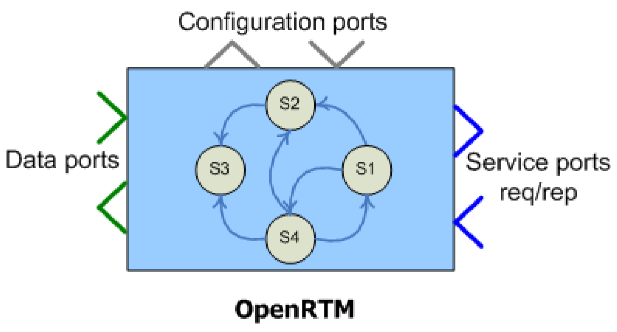


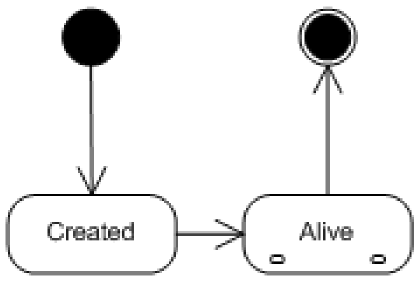
Figure 2.4: The OpenRTM component implementation model.

The application programmer point of view: As has been previously mentioned, an RT-Component is a passive functional unit that does not have a thread of execution. It is implemented as a class which is inherited from a special base class defined in the RT-Component framework. All the required functionality of a component is provided by overriding methods of this base class. From the perspective of a component's life cycle dynamics, any component in the OpenRTM framework goes through a life cycle consisting of a sequence of states ranging from component creation via execution to destruction (see Figure 2.5) [3, 8, 9]. These states can generally be categorized as:

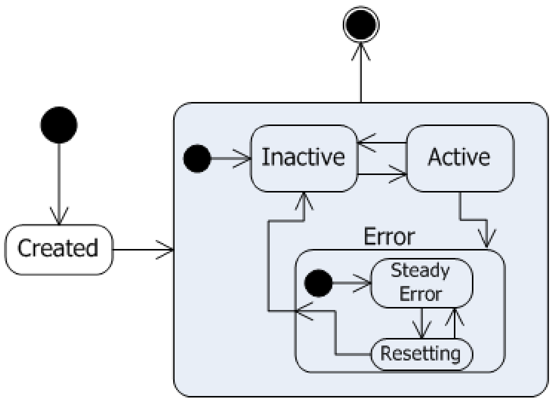
- The *Created* State (Created)
- The *Alive* State (Alive), (The alive state itself has multiple states inside, which is explained below) and
- The *Final* State

Since the component functionality is basically defined by methods of a single class, the component creation process is almost the same as the instantiation of an object from a class in object-oriented software framework, albeit with some additional peculiarities. That is, an RT-Component is created by a manager (RTC Manager) and the manager manages the life cycle of the RT-Component. Concretely, the manager calls the `onInitialize` function after it creates an instance of RT-Component. It also calls the `onFinalize` function when RT-Component finalizes. In this way, RT-Component is programmed by describing necessary details in each processing allocated in specific timing of the RT-Component's life cycle (this processing is referred to as an *Action*). This is to some extent similar to the process of initialization of LAAS/GenoM modules where the developer also defines an initialization step in the Genom description language [12, 13, 14, 15, 16, 17].

The first time an RT-Component is created it is in `Created` state. Afterwards, it transits to `Alive` state and a thread of execution is attached to it (one thread per component). This thread is referred to as an execution context of a component. Within this context all the states and cycle times are defined. We briefly examine a component's execution dynamics as depicted in Figure 2.5. OpenRTM component statemachine can be viewed from two perspectives. From component container (execution context/thread related) point of view and component's own point of view. Execution context defines two states, `Stopped` and `Running`. Naturally, all the necessary computation of the component takes place when its thread is in the `Running` state 2.5. The execution context reacts with the appropriate callbacks upon arrival of respective events. Initially, the component is in `Stopped` state and upon arrival of a start event it transits to `Running` state. When the component is in `Running` it can be in one of the `Inactive`, `Active` and `Error` states of the `Alive` super state 2.5, [3].



(a) simplified



(b) expanded

Figure 2.5: The OpenRTM component state machine model[3].

Immediately after an RT-Component is created, it will be in the `Inactive` state. Once it is activated, the `onActivate` callback is invoked, and the component transits to the `Active` super state. While in this state, the `onExecute` action is executed periodically. A component's main processing is usually performed within this method. For example, controlling a motor based on the data from other components. The component will stay in the active state until it is deactivated or transits to the `Error` state. In either case, appropriate callbacks are invoked. When in the `Error` state, and until being reset externally, the `onError` method will be invoked continuously. If the `onReset` method is successful, the RT-Component goes back to the `Inactive` state. A problem here is that if resetting fails, a component may stay in the `Error` state for an indefinite period of time. There should be some sort of mechanism to take care of this, for instance automatically resetting the component after a fixed period of time or timeout. Considering this model, ideally the main task of a component developer should be overriding each of the state-associated methods/callbacks of the template component with the desired functionality

[3, 8, 9, 10, 11].

OpenRTM comes with an automatic component generation tool, RtcTemplate. RtcTemplate can generate two types of components based on the provided input. By default, when only a module name is given as an argument, it generates data-driven components with data ports only. In case it is invoked with an **.idl* component description file as argument, it generates code for service port support. This is basically done via a wrapper around CORBA IDL compiler, therefore it generates a similar set of files: stubs and skeletons. This is valid for component with service ports because they should know their peers in advance.

The component generation process is to some extent similar to that of the LAAS/GenoM system [12, 13, 14, 15, 16, 17]. In LAAS/GenoM, the developer has to describe the component model, including its execution states, the data shared with other components, the component execution life cycle, and runtime features such as threads etc. This information is then provided to the `genom` command line tool, which generates skeletons for the components in the required programming language. In GenoM, the implementation language is mostly C, whereas RtcTemplate supports C++, Python and Java [18, 19, 3].

In addition to the problem of development of functional components, there is also the problem of composition, also referred to as "system integration". How to connect components with each other without extra overhead of writing special code for it? OpenRTM takes care of this issue by introducing RtcLink, an Eclipse-based graphical composition tool. It provides a basic operating environment required in order to build a system by combining multiple RT-Components. It also enables the acquisition of meta-information of RT-Components by reading the profile of a component and its data and service ports. The other alternative to provide the functionality of RtcLink would be to implment an editing program for connecting components.

The OpenRTM/AIST distribution model is based on distributed objects. It is based on the omniORB implementation of the CORBA specification. In RtcLink, OpenRTM allows to define a subscription type when connecting components (what data the component should receive): *flush*, *new*, *periodic*, *triggered*. Additionally, the developer can define both a communication policy, either pull or push mode, and a communication mechanism, either CORBA or TCP sockets. An OpenRTM component's computation model (the core logic) allows to perform any computation/processing need as long as they can be defined via the provided callback methods tied to the state. It is also possible to combine OpenRTM with external software packages, such as Player/Stage. The build process is also simplified because both OpenRTM components and Player/Stage do come in the form of dynamic libraries. Listing 3 shows the RT-Component structure as it is implemented in OpenRTM/AIST.

```
class RTObject_impl
: public virtual POA_RTC::DataFlowComponent,
public virtual PortableServer::RefCountServantBase
{
  public:
   RTObject_impl(Manager* manager);
   RTObject_impl(CORBA::ORB_ptr orb, PortableServer::POA_ptr poa);
   virtual ~RTObject_impl();

  protected:
   virtual ReturnCode_t onInitialize();
   virtual ReturnCode_t onFinalize();
   virtual ReturnCode_t onStartup(RTC::UniqueId ec_id);
   virtual ReturnCode_t onShutdown(RTC::UniqueId ec_id);
   virtual ReturnCode_t onActivated(RTC::UniqueId ec_id);
   virtual ReturnCode_t onDeactivated(RTC::UniqueId ec_id);
   virtual ReturnCode_t onExecute(RTC::UniqueId ec_id);
   virtual ReturnCode_t onAborting(RTC::UniqueId ec_id);
   virtual ReturnCode_t onError(RTC::UniqueId ec_id);
   virtual ReturnCode_t onReset(RTC::UniqueId ec_id);
   virtual ReturnCode_t onStateUpdate(RTC::UniqueId ec_id);
   virtual ReturnCode_t onRateChanged(RTC::UniqueId ec_id);
```

```
    public:
     virtual ReturnCode_t initialize()
       throw (CORBA::SystemException);
       ...
  };
```

Listing 3. The OpenRTM component signature.

2.1.3 Player

Player is a software package developed at the University of Southern California. The main objective of Player is to simplify the access to hardware resources and devices for higher-level or client robotics modules and applications (e.g. localization, path planning, etc). Player can be viewed as an application server interfacing with robot hardware devices and user-developed client programs.

Figure 2.6 depicts a player-based application consisting of server space and client space code. In client space an application developer uses player client libraries and APIs to access the hardware resources in server space. All the communication between clients and the player server takes place through message queues. A message queue is associated with each player device. Devices themselves are in some sense an aggregation of hardware device drivers and interfaces for accessing the driver functionality. Therefore, anything incoming/outgoing into message queues is then relayed to drivers, which take care of publishing data or subscribing to data through a queue of the device they are part of. Note also, that in addition to physical hardware devices, the player server allows to interface with virtual devices, which can exist on simulated robot platforms. These virtual platforms may be present either in Stage (2D) or in Gazebo (3D) graphical simulators [20, 21, 22, 23, 24]. Since the Player server functions as an application

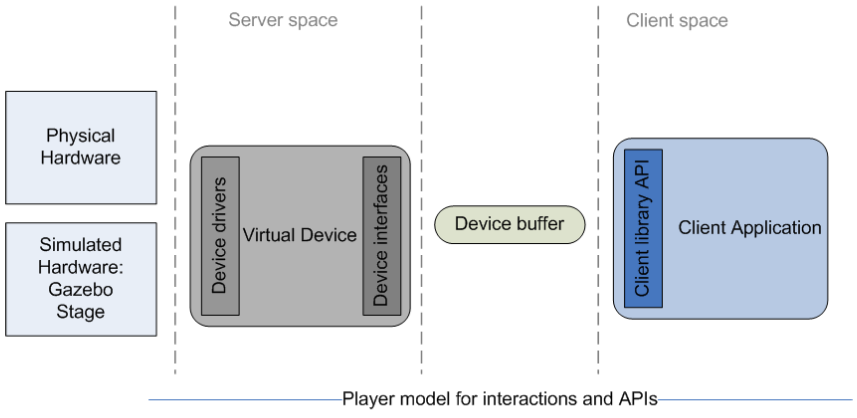


Figure 2.6: The Player server implementation model.

server all the device functionality it presents is either precompiled into it or loaded as plugins. These plugins/precompiled drivers are of shared object format (.so) and are device factories. In order to instantiate particular devices the developer needs to indicate them in the Player server configuration file (.cfg file). This file contains such information as driver name, interface types of this driver, device parameters, etc. Listing 4 shows an excerpt of such a configuration file as taken from an example Player installation.

```
driver
(
  name "urglaser"
  provides ["laser:0"]
```

```
    port "/dev/ttyS0"
    #port "/dev/ttyACM0"
    pose [0.05 0.0 0.0]
    min_angle -115.0
    max_angle 115.0
    use_serial 1
    baud 115200
    alwayson 0
  )
  driver
  (
    name "vfh"
    provides ["position2d:1"]
    requires ["position2d:0" "laser2d:0"]
    cell_size 0.1
    window_diameter 61
    sector_angle 1
    safety_dist 0.3
    max_speed 0.3
    max_turnrate_0ms 50
    max_turnrate_1ms 30
  )
```

Listing 4. Illustration of Player configuration file internals.

This device-related information is described by special keywords which are part of the Player interface specification. The most important ones of these are *name*, *requires*, and *provides*. The *name* keyword serves as an identifier for the Player server and tells which driver should be instantiated. After the server has instantiated a driver to make it accessible to clients, this driver should declare its interfaces. These interfaces could have `required` or `provided` polarities. For a complete reference of configuration keywords check the Player interface specification [25].

The application programmer point of view: To understand the inner workings of Player it is helpful to look into how the server knows how to choose the indicated driver from the precompiled/plugin drivers and instantiate it. Additonally, we will try to show how messages from client space are delivered to the right device.

When the Player server is launched with a configuration file specifying the underlying hardware resources, the first thing it does is to check a driver name in the driver table and register it. The driver table is kept inside the Player server. This procedure is valid when the driver is part of the player core. If driver is in the form of a plugin/shared library, then the name of the shared library which implements the driver needs to be provided. After this process, the Player server reads the information about the interface which is also in the same declaration block in the configuration file (see Listing 4). As illustrated by the listing, there is a special syntax to declare an interface for the driver. This syntax provides information on how many instances of this particular interface should be created (for details, see [25]). Then the type of the interface is matched with appropriate interface CODE (ID) to create that interface. This is needed because each interface has its own specific data and command message semantics. That is, there is a predefined list of data structures and commands that each interface can communicate [25]. Each interface specific message consists of the following headers:

- Relevant constants (size limits, etc.)
- Message subtypes
  - Data subtypes - defines code for a data header of the message
  - Command subtypes - defines code for a command header of the message
  - Request/reply subtypes - defines code for request/reply header of the message

- Utility structures - defines structures that appear inside messages.
- Message structures
  - Data structures - defines data messages that can be communicated through this interface.
  - Command structures - defines command messages that can be communicated through this interface.
  - Request/reply structures - defines request/reply messages that can be communicated through this interfaces.

Listing 5 provides an overview of the message consituents for a *localization* interface; the excerpt is taken from a Player manual [25, 23].

```
#define  PLAYER_LOCALIZE_DATA_HYPOTHS   1
Data subtype: pose hypotheses.
#define  PLAYER_LOCALIZE_REQ_SET_POSE   1
Request/reply subtype: set pose hypothesis.
#define  PLAYER_LOCALIZE_REQ_GET_PARTICLES   2
Request/reply subtype: get particle set.
typedef player_localize_hypoth  player_localize_hypoth_t
Hypothesis format.
typedef player_localize_data  player_localize_data_t
Data: hypotheses (PLAYER_LOCALIZE_DATA_HYPOTHS).
typedef player_localize_set_pose  player_localize_set_pose_t
Request/reply: Set the robot pose estimate.
typedef player_localize_particle  player_localize_particle_t
A particle.
typedef player_localize_get_particles  player_localize_get_particles_t
Request/reply: Get particles.
```

Listing 5. An example Player interface specification.

Based on the discussion above one can consider the Player server a real component-based software system. Component is used as a relative term here and refers to an entity with its own execution context and interaction endpoints. A more thorough definition of a component is given in Section 2.2. The Player server can be considered as a component with three different interface semantics based on the various type of information that is communicated to and from it [25]. These are `DATA`, `COMMAND`, and `REQUEST/REPLY`. Listing 6 shows possible message subtype semantics and their respective codes.

```
#define   PLAYER_MSGTYPE_DATA   1
A   data   message.
#define  PLAYER_MSGTYPE_CMD 2
A command message.
#define PLAYER_MSGTYPE_REQ 3
A request message.
#define PLAYER_MSGTYPE_RESP_ACK 4
A positive response message.
#define  PLAYER_MSGTYPE_SYNCH 5
A synch message.
#define PLAYER_MSGTYPE_RESP_NACK 6
```

Listing 6. Example of various Player message subtypes.

In Player, clients can use two modes for data transmission: `PUSH` and `PULL`. These modes affect a client's message queues only, i.e. they do not affect how messages are received from clients on the server side. In `PUSH` mode all messages are communicated as soon as possible, whereas in `PULL` mode the following holds[25]:

- A message is only sent to a client when it is marked as ready in client's message queue.
- `PLAYER_MSGTYPE_DATA` messages are not marked as ready until the client requests data.
- `PLAYER_MSGTYPE_RESP_ACK` and `PLAYER_MSGTYPE_RESP_NACK` message types are marked as ready upon entering the queue.
- When a `PLAYER_PLAYER_REQ_DATA` message is received, all messages in the client's queue are marked as ready.

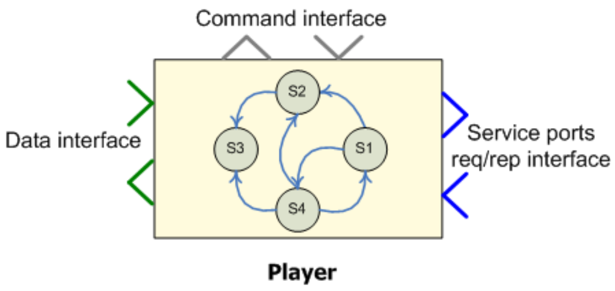


Figure 2.7: Player component model.

The Player driver/plugin can be considered to be a component in the given context, because it is reusable by other player software, has strictly predefined interfaces, provides a functionality, is in binary form and can be composed with other drivers/plugins to provide more complex functionality to clients. As mentioned in the paragraph above these drivers interfaces can cope with `DATA`, `COMMAND`, and `REQUEST/REPLY` semantics. Here the difference between `COMMAND` and `REQUEST/REPLY` is that in the latter client and server need to synchronize through `ACK` or `NACK` messages. Therefore, the Player driver can be depicted as in Figure 2.7.

```
class Driver
{
  private:
  ...
  protected:
  ...
  public:
  ...
   virtual int Setup() = 0;
   virtual int Shutdown() = 0;
   virtual void Main(void);
   virtual void MainQuit(void);
   ....
};
```

Listing 7. An example of a Player driver signature.

All Player drivers/plugins have their own single thread of execution. Listing 7 describes the `Driver` class definition and its life cycle management method prototypes. The Player driver goes through three main states during its life cycle:

- *Setup*: In this state driver is initialized and is active when the first client subscribes.
- *Shutdown*: In this state driver finishes execution and is active when the last client unsubscribes.
- *Main*: In this state, the driver performs its main functionality. After execution finished, the thread exits the *Main* state and goes into the auxiliary *MainQuit* state, which performs additional cleanup when the thread exits.

2.1.4 ROS

The Robot Operating System (ROS) is software developed at the start-up company Willow Garage. Literally, ROS is not a component-oriented software as defined in [26]. It does not define any specific computation unit model with interaction endpoints as it has been in other systems considered so far (sections 2.1.1 and 2.1.2). But like in many programming paradigms (functions in functional programming, objects in object-orientation etc), ROS also strives to build applications from 'modular units'. In the ROS programming model, the modular programming unit is a *node*. The *node* can be considered as a block of functional code performing some sort of computation in a loop. The results of these computations are not available to external parties as long as there is no an interaction endpoint attached to the node. ROS defines two types of such interaction endpoints. The distinction is made with respect to the *semantics* of information communicated through an endpoint and *how* those pieces of information should be communicated. Before delving into details of each endpoint type, one needs to emphasize that in ROS all the necessary information exchange among nodes is performed through *messages*.

Messages are strictly typed data structures. There can be simple data structure and compound data structure messages. Compound messages can be formed by nesting other arbitrarily complex data structures. In order to achieve platform independence (here platform means the implementation language) as well as to facilitate definition of user defined data types, all messages in ROS are described using the `Message Description Language`. Message specification is stored in `.msg` text files which basically consist of two parts, *fields* and *constants*. Fields represent the data type. In addition to a simple *field:constant* pair, messages may have a *header*, which provides meta-information about the message. Like in other systems already reviewed and in interacting software systems in general, communicated data types are part of the `interface contract` between a producer and a consumer. That is, in ROS for nodes to understand each other they need to agree on the message types. Listing 8 below shows a very simple description file for a multi-dimensional array of 32-bit floating point numbers; in this listing MultiArrayLayout is also a message specified in another description file.

```
MultiArrayLayout  layout
float32[]         data
```

Listing 8. '.msg' file definition.

The following types of endpoints are foreseen in ROS:

- In case of the exchanged information having `data` semantics (e.g. data from sensors) and being communicated mostly asynchronously (non-blocking mode) between invoker and invokee, the endpoint is analogous to the data ports in OpenRTM (see Section 2.1.2) and OROCOS (see Section 2.1.1). In ROS, this functionality is achieved through introduction of the messages and the concept of *topics* to which the messages are published. *topic* serves as an identifier for the content of a message. The publisher node publishes its messages to a particular topic and any interested subscriber node can subscribe to this topic. There can be multiple nodes publishing to the same topic or multiple topics, as well as many nodes subscribing to the data on those topics. Such exchange of data has `one-way semantics` and is the corner stone of a publisher/subscriber interaction model. Usually neither the publishers nor the subsribers are aware of each others existance, i.e. they are completely decoupled from each other and the only away for them to learn about topics/data of interest is through a naming service. In ROS, this functionality is performed by the *master node*. The master node keeps track of all the publishers and topics they publish to. Thus, if any subscriber requires data, the first thing it does is to learn from the master node the list of existing topics. Note that in this three-way interaction model (publisher/master, master/subscriber, publisher/subscriber) no actual data is routed through the master node.

- In case of the exchanged information having `command` semantics and being communicated mostly synchronously (blocking mode) between invoker and invokee, the endpoint is analogous to service ports in OpenRTM (see Section 2.1.2). In ROS, this functionality is achieved through introduction of a *service* concept. The service is defined by a string name and a pair of messages, a `request` and a `reply` message (also check Section 2.1.3 for similar concepts). Analogous to messages, services are described using the *Sevice Description Language*, which directly builds upon the Message Description Language. Therefore, any two messages files concatenated with '- - -' form a service description file. A `.srv` file specifies request and response parameters for a service. As in the case of `.msg`, they may also have *header* containing meta-information about the service. Listing 9 below shows an example of a service description file as it is given in the ROS tutorial [27]. Unlike messages, it is not possible to concatenate services. In ROS this makes sense, because in general two arbitrary operations, which are eventually executed upon a service request, cannot simply be aggregated. Another difference of services compared to publishing of data is that services have `two-way semantics`.

```
int64 a //request parameter1
int64 b //request parameter2
---
int64 sum //response value
```

Listing 9. '.srv' file content.

Based on the discussion above, a ROS "pseudo-component" can be represented as in Figure 2.8. Both publisher/subscriber and request/reply type of interactions among nodes are defined

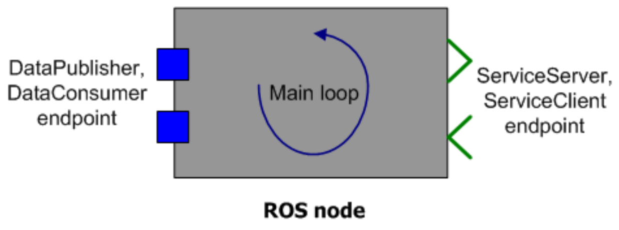

DataPublisher, DataConsumer endpoint
Main loop
ServiceServer, ServiceClient endpoint
ROS node

Figure 2.8: ROS pseudo-component model.

through the XML-RPC high-level messaging protocol. Standard XML-RPC specifies how the messages should be encoded in XML format, what interaction policies communicating parties use to exchange those messages, and what transport protocol to use to transport messages from one party to another [28, 29]. In this process, XML-RPC also takes care of serializing and deserializing message content. Such structuring of XML-RPC messaging is also used in ROS. That is

- The ROS Message Description Language is similar to XML-RPC data model and allows to specify message data types. But unlike XML-RPC, it uses custom text syntax rather than XML.

- The ROS Service Description Language (SDL) is in line with XML-RPC request/response structures. In both one can describe service signatures (i.e. name, parameters, return values etc).

- In terms of tranport protocols, ROS relies on TCP and UDP, so does XML-RPC, though XML-RPC messaging is often embedded in HTML and transported through high level HTTP protocol [28, 29].

The application programmer point of view: In order to utilize ROS functionality a developer needs to program nodes using client libraries, roscpp, rospython. From the deployment point of view, each ROS node is a stand-alone process, thus has its own main loop of execution. Unlike other frameworks, ROS nodes do not define any specific life cycle state machines. This is achieved through the Resource Acquisition Is Initialisation (RAII) interface of a node. The developer just needs to initialize the ROS environment through the `ros::init()` function and create a `NodeHandle()`, which is responsible for instantiating everything necessary for that node (e.g. threads, sockets etc). Listing 10 below shows how a data-driven ROS node looks like in C++; the example is provided in ROS tutorials [27, 30].

```
 1 int main(int argc, char **argv)
 2 {
 3    ros::init(argc, argv, "talker");
 4    ros::NodeHandle n;
 5    ros::Publisher pub = n.advertise<std_msgs::String>("chatter", 1000);
 6
 7    int count = 0;
 8    ros::Rate r(10);
 9    while (n.ok())
10    {
11       std_msgs::String msg;
12       std::stringstream ss;
13       ss << "hello world " << count;
14       ROS_INFO("%s", ss.str().c_str());
15       msg.data = ss.str();
16
17       pub.publish(msg);
18       ++count;
19
20       ros::spinOnce();
21       r.sleep();
22   }
23   return 0;
24 }
```

Listing 10. ROS node in C++.

After allocating the necessary resources to the node (line 4), a data port is attached to it and the data messages from it will be advertized at the topic “chatter” with queue size of 1000 bytes (line 5). These data are then published by invoking `publish()` (line 17).

2.2 Analysis and Comparison of Component Models

2.2.1 Introduction and Essential BCM

The previous section provided general information on some of the current component-based software approaches in robotics. In this section we survey and compare the primary constructs of the component models underlying each system. The comparison will also include some additional software systems which have not been described above). The comparison is performed with respect to the *draft* BRICS component meta-model (BCM)2.9 which we take as a reference and describe below. Note that it does not imply in any way that the BRICS component model is the best choice, it just simplifies the evaluation process, because of the existance of the reference. We start by describing a number of concepts and respective modelling primitives which are used in BRICS component meta-model.

1. **Component**: A component is *"a convenient approach to encapsulate robot functionality (access to hardware devices, simulation tools, functional libraries, etc.) in order to make them interoperable and composable, no matter what programming language, operating system, computing architecture, or communication protocol is used."* [26, 6].

2. **Port**: In the context of BCM, a port is a software component interface (e.g. *scanner2D, position3D*), which groups a set of operations (e.g. *sendScan(), getPos()*) that can only communicate information with *data* semantics to and from other component ports in publisher/subscriber mode (e.g. a set of scan points from a laserscanner, or an array of odometry values from a localisation component). That is, a port represents the data flow of the BCM. Therefore, an inter-component interaction policy is part of the data flow interface and can informally be described by the following expression:

$$BCM\_port = functional\_interface + publisher/subscriber\_interaction\_policy$$

3. **Interface**: In BCM, an interface is a software component interface (e.g. *scanner2D, position3D*) which groups a set of operations (e.g. *setApexAngle(), setResolution()*) that can only communicate information with *command* semantics (e.g setting device/algorithm configurations). In other words, it represents component control flow. In the context of several of the robot software systems presented above, a BCM interface is also referred to as a *service* (see e.g. Section 2.1.2). As in the case of a port, an inter-component interaction policy is part of the BCM interface in the form of a remote procedure call. Informally, this can be expressed as

$$BCM\_interface = functional\_interface + client/server\_interaction\_policy$$

4. **Operation**: An operation is part of the software component interface and is equivalent to a method in OOP (e.g. *setScan(), getConfiguration()*). Different types of operations cen be defined.

5. **Connector and Connection**: A connector is a language or software archictecture construct used to connect components. Since it is a construct, it can be instantiated with different annotations, such as types and policies, e.g. *type → TAO, AMQP, etc; policy → publisher/subscriber, client/server, peer-to-peer, etc*). A connection is the concept that defines whether components are connected through their particular endpoints/ports. Additionally, the connection is a system model-level primitive.

6. **State Machines**: The execution model, i.e. the sequence of statements executed during runtime, is defined by a state machine. BCM actually uses several state machines for well-defined purposes:

- The *component-internal life cyle state machine* defines the phases of the life cycle of a component instance, starting from the initialization of ports, interfaces, and resource allocation, to cleanup and execution of the actual component functionality/algorithm. In BCM, the life cycle state machine consists of three states *init, body, final* and well-defined transitions between them.
- The *functional algorithm state machine* is a second level execution manager — the first level is the component life cycle manager — and runs within the first level *body* state. BCM does currently not define a particular execution model for this level.
- The *interface state machine* defines stateless and stateful interfaces. BCM introduces contracts for the component service interfaces. The contracts are specified as state machines which are part of the service interface and define constraints on the execution order of the operations specified by the interface. Depending on the specification of the state machine component a service interface can be stateful or stateless.

7. **Data Types**: BCM specifies a number of predefined data types for use in communication between components. These include a comprehensive set of primitive data types and a set of commonly used compound data types. The latter conveys that developers can define their own data types which follow particular requirements for component data exchange.

8. **Operation Mode**: This concept relates to the deployment aspect of the component. The component can be either in asynchronous, synchronous or any execution modes.

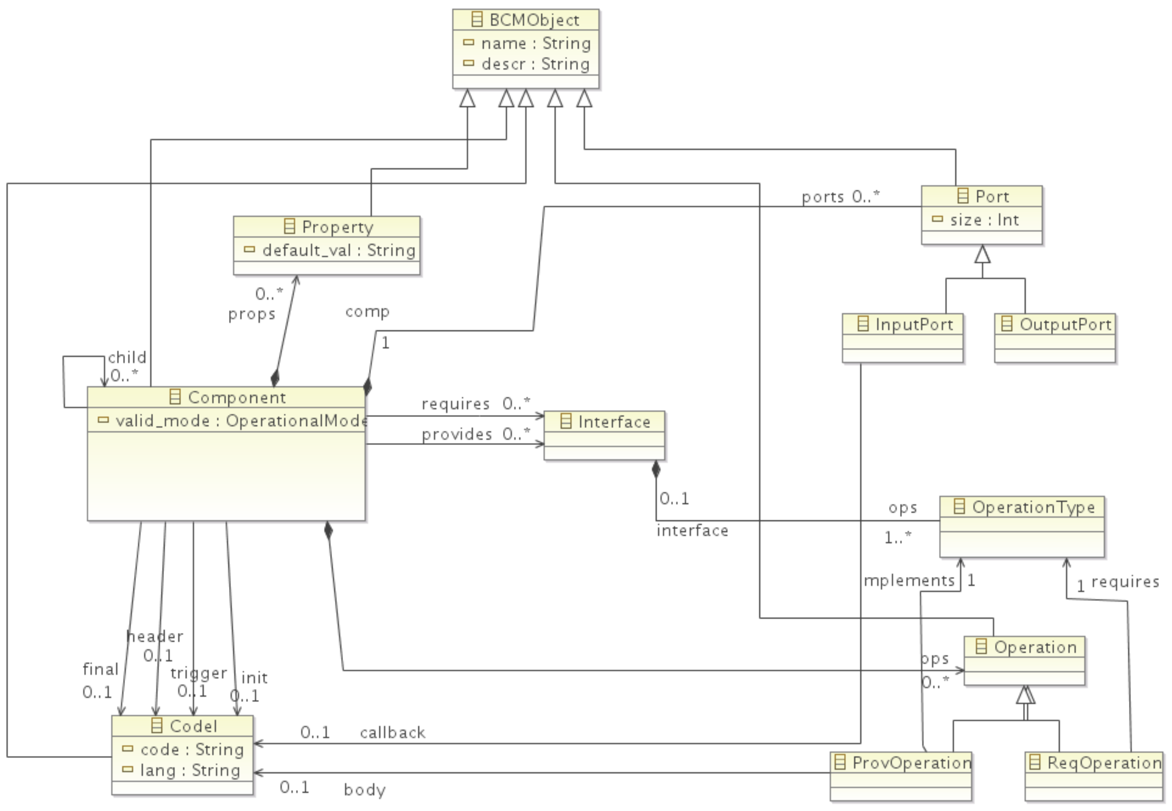


Figure 2.9: Simplified brics component model.

Having briefly explained BCM entities, we introduce a simple evaluation method based on the analysis results of other component models. One of the major objectives of defining BCM and implementing toolchain support around it is to ensure interoperability among component-based robotics frameworks. Therefore the outcome of this evaluation should show how difficult it is to achieve this goal in terms of the existing BCM. The evaluation results are delivered in the form of tables indicating the amount of effort required to implement or map a feature of particular framework in terms of BCM entities. To keep initial process simple there have been three levels of effort defined, which are:

- $\odot$ - almost no effort to map
- $\ominus$ - average effort to map
- $\oplus$ - considerable effort to map

At the same time only coordination and communication primitives of the models are compared. We would like to emphasize that this evaluation approach is not purely objective, because it requires one to know and be experienced with technical details of each framework.

BCM	**Port**	**Interface**	**State Machine**	**Connection**
SW system name	$\odot$	$\ominus$	$\oplus$	$\odot$

Table 2.1: Mapping between BRICS CMM constructs and other systems.

2.2.2 ORCA2

1. Port: First of all, ORCA2 component interfaces follow the same template, i.e. they consist of mostly the same methods. There is no clear distinction on the model level between data and control flow. When analyzed on the code level, a component's *provides* and *requires* data interfaces can be considered equal to ports in BCM. But unlike BCM, ORCA2 does not support explicit port constructs. ORCA2 *provides* and *requires* data interfaces all communicate also using a publisher/subscriber policy.
2. Interface: Similar situation as for ports above. Though, ORCA2 implicitly defines interfaces as first class entities, since it also uses the *Slice* interface defition language of the ICE middleware to define its components.
3. Operation: There is no explicit distinction between operations in a component interface. Additionally, there is no such concept as operation type in the context of ORCA2 components.
4. Connection: ORCA2 does not feature any explicit constructs to represent connections. In most cases system builders manually indicate system topology in a file. Information in this file consists of a component name, the portID it is listening to, the type of the protocol, and the *requires* and *provides* interfaces of the component.
5. State Machines
   - Component Internal Life Cyle State Machine: ORCA2 implicitly structures the component life cycle across two states, *start* and *stop*, which are semantically equivalent to the *init* and *final* states in the BCM state machine. Thus, the BCM life cycle manager can be mapped with minor modifications to a ORCA2 CM life cycle manager. Here *start* and *stop* are only responsible for initialization and cleanup of component resources, e.g. the component's main thread is created in the *start* state.

- Functional Algorithm State Machine: The ORCA2 component functionality is structured in *initialise, work, finalize* states. A main thread which is created in the component level *start* state is attached to this state machine. This can be considered to be a thread context state machine. There is also a predefined state machine for driver implementation, which consists of *warning, faulty, Ok* states.
- Interface State Machine (stateless and stateful interfaces): There is no explicitly defined interface level state machine. All interface by default are stateless (or implicitly/predefined stateful)

6. Data Types: The form of the data types is predefined for each interface. Also, ORCA2 follows *BROS* guidelines[] on data representation for kinematics and dynamics (e.g. data structures for velocity, position, force ect).
7. Operation Mode: ORCA2 relies on the ICE middleware (IceUtil:Thread) threading, so one needs to set the appropriate mode through Ice runtime if it is possible, otherwise Ice schedules the threads itself.

Based on the evaluation above we can conclude on how much effort it will require to instatiate ORCA2 CM based on BCM.

BCM	**Port**	**Interface**	**State Machine**	**Connection**
ORCA2	$\odot$	$\odot$	$\oplus$	$\ominus$

Table 2.2: Mapping between BCM constructs and ORCA2.

2.2.3 OpenRTM

1. Port: OpenRTM components have similar constructs with the same semantics and similar contexts as defined in BCM. OpenRTM CM includes *port* as stand-alone construct. Ports are unidirectional as in BCM and are used to transfer data in publisher/subscriber mode. In OpenRTM, components that have only data ports for interaction are referred to as data flow components. OpenRTM allows to define a polarity (*required, provided*) for a port.
2. Interface: As in BCM, an interface in OpenRTM represents mostly control flow semantics and is referred to as a service port. It transfers information in client/server mode as in BCM. It is defined as a separate construct.
3. Operation: A concept of operation or operation type is not explicitly specified in OpenRTM. Although it provides a configuration interface in addition to data and service ports.
4. Connection: OpenRTM models inter-component interactions through the connectors which are part of the framework. Connectors are specified through their connector profiles which contain a connector name, an id, the ports it is connecting, and additional attributes. There is no explicit support for connector objects. Therefore they are implicitly defined by the connections between components.
5. State Machines: As in BCM, OpenRTM also distinguishes between functional (`core logic` in OpenRTM terminology) and container contexts on the model level, thereby defining different execution models. But OpenRTM does not describe explicit hierarchical relations, neither on the model level nor on the code level between these state machines. This sometimes lead to confusion.

- Component Internal Life Cyle State Machine: This is part of the container context. OpenRTM defines a state machine consisting of `created, alive, final` states. This specifies whether a component was created and with what resources, parameters and whether it is waiting for an *activity* — this term is defined in the context of OpenRTM and has similar semantics as a task) — to execute.
- Functional Algorithm State Machine: This is part of the core logic of the component and is related to the execution context or thread state machine, which is composed of `stopped` and `running` states. As soon as the component is in *running* state, the core logic will be executing according to state sequence `inactive, active, error`.
- Interface State Machine (stateless and stateful interfaces): There is no explicit specification of interface protocol in OperRTM. By default, all the interfaces are stateless (or implicitly/predefined stateful).

6. Data Types: OpenRTM relies on UML as well as IDL primitive types. It also defines type `Any`. There are no interface-specific data types (e.g. `forceVector, positionVector` etc) as in systems such as Player and ORCA2.

7. Operation Mode: OpenRTM allows the definition of different thread execution modes as in BCM. This is referred to as execution type. There are *periodic, event driven, other* modes. So there is a one-to-one match with BCM operation modes.

BCM	**Port**	**Interface**	**State Machine**	**Connection**
OpenRTM	$\odot$	$\odot$	$\oplus$	$\ominus$

Table 2.3: Mapping between BRICS component model constructs and OpenRTM.

2.2.4 GenoM

1. Port: The concept of port does not exist in its given interpretation in GenoM modules. In order to exchange data, GenoM modules use *posters*, which are sections of shared memory. There are two kinds of posters: `control posters`, which contain information on the state of the module, running services and activities, client IDs, thread periods, etc, and `functional posters`, which contain data shared between modules (e.g. sensor data). Additionally, ports in BCM have publisher/subscriber semantics and are transmitted in asynchronous mode, whereas in GenoM all communication has synchronous semantics [14, 15, 16, 17].

2. Interface: GenoM modules use a *request/reply* library interface to interact with each other, which is semantically equal to *provides/requires* services. There can be two types of requests: `control` requests and `execution` requests. They are both passed over to the module controller thread and the execution engine, respectively.

3. Operation: Operations as defined in BCM are not existent, but it is possible to relate them to the *request/reply* interface of the module.

4. Connection: Connections do not exist as a separate entity. The developer needs to specify in model description file for each model which shared memory section it needs to have access to. On the *request/reply* interface level connections are set up through a TCL interpreter, which plays the role of an application server to all running modules. The developer writes a TCL script in which he defines the module connections. This is very similar to the Player approach, where the role of the TCL interpreter is taken by the Player device server [12, 13].

5. State Machines:
   - Component Internal Life Cyle State Machine: This is defined in a module description file within the context of a module execution task. It is not predefined in the framework itself, so that the developer is free to define his own life cycle manager.
   - Functional Algorithm State Machine: Each request, regardless of its type (control or execution), has its own state machine which can be defined in the module description file. The form and size of these state machines are restricted to be a subset of the super-state machine which is predefined in the system and consists of 6 states, *START, EXEC, SLEEP, END, FAIL, ZOMBIE, INTER, ETHER*, and transitions between them. When an *activity* is in one of these states, a function associated with this state and known as a *codel* is performed. Since request-associated state machines are subsets of the predefined super-state machine, a user is not required to implement codels for the states that he considers not necessary.
   - Interface State Machine (stateless and stateful interfaces): This does not exist, and, by default, a *request/reply* interface is stateful and a functional poster interface (data interface) is stateless.

6. Data Types: GenoM does not support any interface-specific data types, but supports both simple and complex data types as they are defined in the C programming lanaguage.

7. Operation Mode: GenoM allows to specify a thread context (referred to as an *execution task context*) with information including periodicity, period, priority, stack size, and life cycle state machine in the module description file. It specifically supports periodic and aperiodic execution modes.

BCM	**Port**	**Interface**	**State Machine**	**Connection**
Genom	⊖	⊖	⊕	⊕

Table 2.4: Mapping between BRICS CMM constructs and GenoM.

2.2.5 OPRoS

1. Port: OPRoS explicitly defines different types of ports. The BCM port concept is semantically equivalent to OPRoS data and event ports. As in BCM, the data is transferred in publisher/subscriber mode. Note, that only non-blocking call are possible through data port.

2. Interface: There is no a construct defining an interface as such in OPRoS, but there is a semantically equivalent port type, the *method* port. Method ports support client/server type of interaction for information exchange. As most of the software systems above, the method interface/port makes up a component's control flow, whereas the data port/event interface makes up the data flow. Additionally, the type of inter-component interaction is part of the interface definition. That is, publisher/subscriber is always associated with data ports, and client/server with method interfaces. Also note that a method port interface can function either in blocking or non-blocking mode.

3. Operation: There is no explicit construct describing operations of the interface. Interface operations can be blocking or non-blocking as specified by their synchronization property.

4. Connection: Only method ports which have matching method profiles can interact with each other.

5. State Machines:

   - Component Internal Life Cyle State Machine: This state machine is managed by the OPRoS component container. All components follow the same predefined life cycle state machine, which is composed of 6 states, *CREATED, READY, RUNNING, ERROR, STOPPED, SUSPENDED*, and the transitions between them.
   - Functional Algorithm State Machine: There is no explicit state machine for algorithm execution.
   - Interface State Machine (stateless and stateful interfaces): There is no explicit state machine or any other means to describe stateful and stateless interfaces.

6. Data Types: Not foreseen to be specified.

7. Operation Mode: Not foreseen to be specified.

BCM	**Port**	**Interface**	**State Machine**	**Connection**
OPRoS	$\odot$	$\odot$	$\oplus$	$\oplus$

Table 2.5: Mapping between BRICS CMM constructs and OPRoS.

2.2.6 ROS

1. Port: ROS nodes have constructs with similar semantics and context as ports defined in BCM. These constructs are data publishers and subscribers that are attached to the node. For details refer to 2.1.4.

2. Interface: ROS services are analogous to interfaces. As in the case of publisher and subscriber, services are not part of the node, but are defined with its context. They provide two-way communication semantics 2.1.4.

3. Operation: Does not exist in a form similar to BCM.

4. Connection: Does not exist in a form similar to BCM.

5. State Machines: Does not exist in a form similar to BCM.

   - Component Internal Life Cyle State Machine: Does not exist in a form similar to BCM.
   - Functional Algorithm State Machine: Does not exist in a form similar to BCM.
   - Interface State Machine (stateless and stateful interfaces): Does not exist in a form similar to BCM.

6. Data Types: A predefined set of robotics-specific and standard data types exists. The developer can also define custom data structures using a Message Description Language 2.1.4.

7. Operation Mode: Does not exist in a form similar to BCM.

BCM	**Port**	**Interface**	**State Machine**	**Connection**
ROS	⊙	⊙	⊖	⊖

Table 2.6: Mapping between BRICS CMM constructs and ROS.

2.2.7 OROCOS

1. Port: OROCOS components have similar constructs used with the same semantics and similar context as in BCM.
2. Interface: The BCM interface is equivalent to an OROCOS method.
3. Operation: Each of the interface types defined in OROCOS is a grouping of operations. These operations are equivalent to their BCM couterparts.
4. Connection: OROCOS provides an explicit connection concept between components.
5. State Machines:
   - Component Internal Life Cycle State Machine: An OROCOS state machine is composed of three main states (see 2.1.1), which can be mapped onto the BCM life cycle state machine.
   - Functional Algorithm State Machine: In OROCOS the developer can define custom state machines which are executed for instance when an event arrives at an event port.
   - Interface State Machine (stateless and stateful interfaces): There is no explicit separation of stateless and stateful interfaces.
6. Data Types: OROCOS provides a set of predefined standard data types as in BCM. One can also create custom data types.
7. Operation Mode: The OROCOS component operation modes are defined through *activities*. There are *periodic*, *sequential*, and *non-periodic* (event-driven) activity types. In BCM, the component execution mode is specified through annotations. Thus, it is possible to map between BCM and OROCOS without any problems.

BCM	**Port**	**Interface**	**State Machine**	**Connection**
OROCOS	⊙	⊙	⊙	⊙

Table 2.7: Mapping between BRICS CMM constructs and OROCOS.

2.2.8 Summary of Comparative Analysis

The results of the analysis can be summarized in the following table:

2.3 Conclusions

This chapter provided an almost exhaustive review of component-based software frameworks in the robotics domain. An attempt was made to objectively analyze and evaluate these robotics component-based software frameworks. The evaluation was performed with respect to BRICS Component Model (BCM), which is currently under development. The final goal of BCM is to incorporate the best practices and most important features from the existing systems and allow

BCM	**Port**	**Interface**	**State Machine**	**Connection**
ORCA2	$\odot$	$\odot$	$\oplus$	$\ominus$
OpenRTM	$\odot$	$\odot$	$\oplus$	$\ominus$
Genom	$\ominus$	$\ominus$	$\oplus$	$\oplus$
OPRoS	$\odot$	$\odot$	$\oplus$	$\oplus$
ROS	$\odot$	$\odot$	$\ominus$	$\ominus$
OROCOS	$\odot$	$\odot$	$\odot$	$\odot$

Table 2.8: Comparison of component modelling primitives in different robot software systems with respect to the BRICS component model.

component level interoperability. This work will allow to justify design decisions behind BCM. During the evaluation process the following points were observed:

- There is not only a zoo of robot software frameworks, but also a zoo of different component models.
- Most of these component models have almost the same type of component interfaces, i.e. data ports and service ports.
- Most component models lack an explicit concept of connectors.
- Most component models have similar finite state machine categories for life cycle management.
- Most component models come in the form of a framework and there is not much tool support provided for application design and testing. Exceptions to this situation are OPRoS, OpenRTM, and ROS, which provide software management and design tool chain.
- Even though most of the components have common features and attributes, there is no systematic approach to reuse software across the systems. Observing current trends in robotics software development, it is realistic to expect that the number of new software packages will grow in the future. This situation is similar to the operating systems domain a decade ago, when there were a handful of systems which then grew in number. Most of those systems provided some means for interoperability among each other. A similar approach should be taken in the robot software domain. Since there is an abundance of robot software systems and component models out there with largely the same functionalities, and at the same time there is no way to persuade people to use *The Grand Unifying Solution*, the best approach to make progress is to achieve interoperability between existing systems on different levels, i.e. on component level, model level, algorithm level, etc.

Chapter 3

Communication Middleware

Makarenko et al.[31] analysed different robotic software frameworks, like Player[32], Orocos[7], and YARP[33], and concluded that all of these frameworks are faced with distributed communication and computation issues. Due to the distributed nature of todays robotic systems this is not surprising. For instance *Johnny*, the 2009 world champion in the RoboCup@Home league is a fully distributed system, e.g. consumer and producer of laser scans are physically distributed connected via an ethernet network [34]. Having a distributed system, it is necessary to address several problems. Ranging from *How is the information encoded?* to *How is the data translation between different machines managed?*. As in other domains (automotive, aviation, and telecommunications), these problems in robotics can be solved by making use of middleware technologies. Besides solving the aforementioned challenges, these middleware technologies hide most of the complexities of distributed systems programming from the application developer. Interestingly, in robotics there is no single middleware technology exclusively used. In fact, every robotic framework comes with it's own middleware technology, which is either handcrafted[32] or an external package integrated into the robotics software framework[31]. However, the decision criteria for which middleware technology to choose are quite often quite fuzzy and not well-defined. Taking this into account, this section attempts to assess middleware technologies in an unambiguous manner, and serves as a screening of the current marketplace in communication middleware technologies.

The remainder of this section is structured as follows. Section 3.1 screens the current marketplace of communication middleware technologies and compiles and categorizes an exhaustive list of relevant technologies. In Section 3.2, the technologies are distinguished between specifications and implementations, briefly described, and compared with respect to some previously defined characteristics. Section 3.3 draws conclusions and gives some recommendations.

3.1 The Marketplace of Communication Middleware

The current marketplace for middleware technologies is confusing and almost daunting[1]. However, several authors already surveyed the current marketplace and tried to categorise the available spectrum of middleware technology. One feasible approach to categorise different middleware technologies is by their means of communication. In [35], Emmerich reviewed state-of-the-art middleware technologies from a software engineering point of view. For doing this, Emmerich introduced the following three middleware classes.

- **Message-oriented Middleware (MOM):** In a message-oriented middleware, *messages* are the means of communication between applications. Therefore, message-oriented middle-

[1]Typing *middleware* in a google search resulted in over 6.390.000 links on January 5th, 2010.

ware is responsible for the message exchange and generally relies on asynchronous message passing (fire and forget semantics) leading to loose coupling of senders and receivers.

- **Procedure call-oriented Middleware (POM):** All middleware which mainly provides remote procedure calls (RPC) is considered as procedure call-oriented middleware. Procedure call-oriented middleware generally relies on synchronous (and asynchronous) message-passing in a coupled client/server relation.

- **Object-oriented Middleware (OOM):** Object-oriented middleware evolved from RPC and applies those principles to object orientation.

Within this survey, these classes were used to classify the middleware technologies from Table 3.1. The classification is shown in the third column. The list has been compiled from various sources including scientific articles from of the robotics domain. The first column labels the name of the middleware technology. The name refers to a concrete implementation (e.g. **mom4j**) and not to a specification (e.g. **mom4j** is an implementation of the **Java Message Service (JMS)** specification). In the second column the license under which the middleware technology is available is mentioned. Please note that the entry *Comm.* means that the middleware solution is commercial. The acronyms *APL*, *GPL*, *LGPL* and *BSD* are referring to the well-known open-source licenses Apache License (APL), GNU General Public License (GPL), GNU Lesser General Public License (LGPL) and BSD License (BSD)[2]. The middleware **Spread** comes with it's own license, which is slightly different from the *BSD* license. The MPI library **Boost.MPI** also comes with it's own license. **ZeroMQ**, **omniORB** and **TIPC** are available under a dual license, which means that some parts (e.g. tools in **OmniORB** (e.g. for debugging) are under GPL and the middleware itself is under LGPL of the projects have different license conditions. Due to the fact that **MilSoft**, **CoreDX**, **ORBACUS** and **webMethods** are commercial solutions they are not really further considered in this survey. Furthermore, the **.NET** platform by Microsoft is not considered as well. **.NET** is still a Microsoft-only solution and therefore not suitable for heterogenous environments as in robotics. In the fourth column of the table the interested reader will find website references for further information.

3.2 Assessment of Communication Middlewares

In the following the middleware technologies (from Table 3.1) are differentiated in associated specifications and implementations, briefly described, and compared with respect to core characteristics such as supported platforms and programming languages (see tables 3.2 and 3.3).

3.2.1 Specification versus Implementation

As mentioned above, Table 3.1 shows only implemented middleware solutions. Some of these implementations are associated to specifications. This means they implement a particular specification. To avoid adulterated assesment (e.g. comparision among specification and implementation) it is necessary to figure out which middleware corresponds to which specification. The breakdown in specification and associated implementation is shown in Table 3.2. In the following paragraphs, specifications and their associated implementations are described (e.g. **Data Distribution Service**) as well as pure implementations without associated specifications (e.g. **TIPC**).

[2]The licenses are available on `http://www.opensource.org/licenses` (last accessed on 05-01-2010)

Middleware	License	Class	Link for more information
OpenSplice	*LGPL*	**MOM**	`opensplice.com/`
MilSoft	*Comm.*	**MOM**	`dds.milsoft.com.tr/en/dds-home.php`
CoreDX	*Comm.*	**MOM**	`twinoakscomputing.com/coredx.php`
Apache Qpid	*APL*	**MOM**	`qpid.apache.org/`
ZeroMQ	*LGPL/GPL*	**MOM**	`zeromq.org/`
webMethods	*Comm.*	**MOM**	`softwareag.com/corporate/default.asp`
mom4j	*LGPL*	**MOM**	`mom4j.sourceforge.net/`
Boost.MPI	*Boost*	**POM**	`boost.org/doc/libs/1_39_0/doc/html/mpi.html`
TIPC	*BSD/GPL*	**POM**	`tipc.sourceforge.net/`
D-Bus	*GPL*	**OOM**	`freedesktop.org/wiki/Software/dbus`
LCM	*LGPL*	**MOM**	`code.google.com/p/lcm/`
Spread	*Spread Lic.*	**POM**	`spread.org/`
omniORB	*LGPL, GPL*	**OOM**	`omniorb.sourceforge.net/`
JacORB	*LGPL*	**OOM**	`jacorb.org/`
TAO	*LGPL*	**OOM**	`theaceorb.com/`
ORBACUS	*Comm.*	**OOM**	`progress.com/orbacus/index.html`
ICE	*GPL*	**OOM**	`zeroc.com/`
XmlRpc++	*LGPL*	**POM**	`xmlrpcpp.sourceforge.net/`

Table 3.1: Compiled list of different middleware technologies

Specification	Middleware
DDS	**OpenSplice**
AMQP	**Apache Qpid**
AMQP	**ZeroMQ**
JMS	**mom4j**
MPI	**Boost.MPI**
CORBA	**omniORB**
CORBA	**JacORB**
CORBA	**TAO**
XmlRpc	**XmlRpc++**

Table 3.2: Middleware specifications and their associated implementations.

Middleware	Java	C/C++	C#	Python	Other
OpenSplice	√	√	√		
Apache Qpid	√	√	√	√	Ruby
ZeroMQ	√	√	√	√	Fortran
mom4j	√				
Boost.MPI		√			
TIPC		√			
D-Bus	√	√	√	√	Perl
LCM	√	√		√	
Spread	√	√		√	Perl and Ruby
omniORB		√		√	
JacORB	√				
TAO		√			
ICE	√	√	√	√	PHP
XmlRpc++		√			

Table 3.3: Middleware implementations and their supported programming languages.

Middleware	Linux	Windows	Other
OpenSplice	√	√	
Apache Qpid	√	√	Java
ZeroMQ	√	√	QNX, Mac OS/X
mom4j	√	√	Java
Boost.MPI	√	√	
TIPC	√		
D-Bus	√	√	
LCM	√	√	Java
Spread	√	√	Solaris, FreeBSD
omniORB	√	√	HP-UX, Solaris
JacORB	√	√	VxWorks, and more
TAO	√	√	Mac OS/X, Solaris
ICE	√	√	Mac OS/X
XmlRpc++	√	√	C++ Posix

Table 3.4: Middleware implementations and their supported platforms.

3.2.2 Data Distribution Service (DDS)

Like CORBA (see Section 3.2.6), the Data Distribution Service is a standard by the OMG consortium. It describes a publish/subscribe service (one-to-one and one-to-many) for data-centric communication in distributed environments. For that reason, DDS introduces a communication model with participants as the main entities. These participants are either exclusive publishers or subscribers, or both. Similar to other message-oriented middleware, publishers/subscribers are sending/receiving messages associated to a specific topic. Furthermore, the standard describes the responsibilites of the DDS as follows:

- awareness of message adressing,
- marshalling and demarshalling, and
- delivery.

The standard itself is divided into two independent layers: the *Data Centric Publish Subscribe* (DCPS) and the *Data Local Reconstruction Layer* (DLRL). The former one describes how data from a publisher to a subscriber is transported according to quality of service constraints. On the other hand the DLRL describes an API for an event and notification service, which is quite similiar to the CORBA event service. Due to the fact that the DDS is independent from any wiring protocol the QoS constraints are dependent on the used transport protocol (e.g. TCP or UDP).

OpenSplice, developed by PrismTech, implements the OMG Data Distribution Service. OpenSplice is available as a commercial and a community edition. The community edition is licensed under LGPL. From a technical point of view, OpenSplice's main focus is on real-time capabilities, and therefore reference applications are in the field where real-time is an important issue (combat systems, aviation, etc.).

3.2.3 Advanced Message Queuing Protocol (AMQP)

Companies like Cisco, IONA, and others specified the Advanced Message Queuing Protocol (AMQP). AMQP is a protocol for dealing with message-oriented middleware. Precisely, it specifies how clients connect and use messaging functionality of a message broker provided by third-party vendor. The message broker is responsible for delivery and storage of the messages.

Apache Qpid: The open source project Apache Qpid implements the AMQP specification. Qpid provides two implementations of the Qpid messaging service. On is a C++ implementation, the other a Java implementation. Interestingly, the Java implementation is also fully compliant with the JMS specification by Sun Microsystems (see Section 3.2.4).

ZeroMQ: The ZeroMQ message-oriented middleware is based on the AMQP model and specification. In contrast to other message-oriented middleware, ZeroMQ provides different architectural styles of message brokering. Usually, a central message broker is repsonsible for message delivering. This centralized architecture has serveral advantages:

- loose coupling of senders and receivers
- lifetime of senders and receivers do not overlap
- broker is resistent to application failures

However, the centralized architecture also has several disadvantages. Firstly, the broker incurs an excessive amount of network communication. Secondly, due to the potential high load of the message broker, the broker itself can become a bottleneck and the single point of failure of the messaging system. Therefore, ZeroMQ supports different messaging models, namely *broker* (as mentioned above), *brokerless* (peer-to-peer coupling), *broker as directory server* (senders and receivers are loosely coupled and they find each other via the directory broker, but the messaging is done peer-to-peer), *distributed broker* (multiple brokers, one each for a specific topic or message queue), *distributed directory server* (multiple directory servers to avoid a single point of failure).

3.2.4 Java Message Service (JMS)

The Java Message Service (JMS)[3] itself is not a full-fledged middleware. It is an Application Programming Interface (API) specified by the Java Consortium to access and use message-oriented middleware provided by third-party vendors. Like in other message-oriented middleware, messages in JMS are a means of communication between processes or more precisely applications. The API itself supports the delivery, production and distribution of messages. Furthermore, the semantics of delivery (e.g. *synchronous*, *transacted*, *durable* and *guaranteed*) is attachable to the messages. Via the Java message interface, the envelope of a message itself is specified. Messages are composed of three parts: header (destination, delivery mode, and more), properties (application-specific fields), and body (type of the message, e.g. serialised object). JMS supports two messaging models: *point-to-point* (a message is consumed by a single consumer) and *publish/subscribe* (a message is consumed by multiple consumer). In the point-to-point case, the destination of a message is represented as a named queue. The named queue follows the first-in/first-out principle and the consumer of the message is able to acknowledge the receipt. In the publish/subscribe case the destination of the message is named by a so called topic. Producers publish to a topic and consumer subscribe to a topic. In both messaging models the reception of messages can be in blocking mode (receiver blocks until a message is available) and non-blocking mode (receiver gets informed by the messaging service when a message is available).

mom4j The open source project mom4j is a Java implementation of the JMS specification. Currently mom4j is compliant with JMS 1.1, but provides downwards compatiblity to JMS 1.0. Due to the fact that the protocol to access and use mom4j is language independent, clients are programmable in different programming languages, e.g. Python and Java.

3.2.5 Message Passing Interface (MPI)

The Message Passing Interface (MPI) provides a specification for message passing especially in the field of distributed scientific computing. Comparable to JMS (see Section 3.2.4), MPI specifies the API but not the implementation. Like Spread and TIPC, MPI introduces groups and group communication in the same manner as the former do. Moreover, the API provides point-to-point communication via synchronous sending/receiving, and buffered sending/receiving. From an architectural point of view, no server or broker for the communication between applications is needed. It is purely based on the peer-to-peer communication model (group communication can be considered as an extension). The communication model of MPI guarantees that messages will not overtake each other.

Boost.MPI Boost.MPI is part of the well-known and popular Boost C++ source library and supports the functionality which is defined in the MPI 1.1 specification.

[3]Specification available at `http://java.sun.com/products/jms/docs.html` (last access 05-01-2010)

3.2.6 Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) is a standard architecture for distributed object-oriented middleware specified by the Object Management Group (OMG)[4]. CORBA was the first specification which heavily applied the broker design pattern in a distributed manner, resulting in providing object-oriented remote procedure calls. Besides remote procedure calls, CORBA specified numerous additional services like naming service, query service, event service, and more.

omniORB implements the CORBA specification and is available for C++ and Python. OmniORB is available under Windows, Linux, and several arcane Unix environments like AIX, UX, and more. From a technical point of view, OmniORB provides a quite interesting thread abstaction framework. A common set of thread operations (e.g. `wait()`, `wait_until()`) on top of different thread implementations (e.g. pthreads under Linux) is available.

JacORB is a CORBA implementation written in Java and available exclusively for Java platforms. From a technical-point of view JacORB provides the asynchronous method invocation as specified in Corba 3.0. Furthermore, JacORB implements a subset of the quality of service policies defined in Chapter 22.2 of the CORBA 3.0 specification. Namely, the sync scope policy (at which point a invocation returns to the caller, e.g. the invocation returns after the request has been passed to the transport layer) and the timing policy (definition of request timings).

TAO is an implementation of the OMG Real-Time CORBA 1.0 specification, developed by Douglas C. Schmidt at Vanderbilt University. The current version of TAO realizes several features of the specification as: real-time ORB, invocation timeouts, real-time mutexes, and more. TAO has been designed for hard real-time applications, however, TAO can be used for every application where an Object Request Broker is needed. One major advantage over other CORBA implementations is the efficient, and more importantly, predictable QoS (Quality of Service) of TAO. TAO has been used not only in robotics (e.g. in MIRO), but also in other domains as in avionics.

3.2.7 XML Remote Procedure Call (XML-RPC)

The XML-RPC protocol uses XML to encode remote procedure calls (XML as the marshalling format). Furthermore, the protocol is bound to http as a transport mechanism, because it uses 'request' and 'response' http protocol elements to encapsulate remote procedure calls. XML-RPC showed to be very well suited for building cross-platform client/server applications, because client and servers don't need to be written in the same language.

XmlRpc++ is a C++ implementation of the XML-RPC protocol. Due to the fact that XmlRpc++ is written in standard Posix C++, it is available under several platforms. XmlRpc++ is some how exotic in this survey. Firstly, the footprint is comparable small. Secondly, no further library is used (XmlRpc++ make use of the socket library provided by the host system).

3.2.8 Internet Communication Engine (ICE)

The Internet Communication Engine (ICE) by the company ZeroC is an object-oriented middleware. ICE evolved from experiences by the core CORBA developers. From a non-technical point

[4]Available at `http://www.omg.org/technology/documents/corba_spec_catalog.htm` (last access 05-01-2010)

of view, the Internet Communication Engine is available in two licenses: a GPL version and a commercial one upon request. Furthermore, the documentation is exhaustive. From a technical point of view, ICE is quite comparable to CORBA. Like CORBA, ICE adapts the broker design pattern from network-oriented programming to achieve object-oriented remote procedure calls, including a specific interface definition language (called Slice) and principles like client-side and server-side proxies. Beyond remote procedure calls, ICE provides a handful of services.

- **IceFreeze** introduces object persistence. By the help of the Berkeley DB states of objects are stored and retreieved upon request.
- **IceGrid** provides the ICE location service. By making use of the IceGrid functionality it is possible for clients to discover their servers at runtime. The IceGrid service acts as an intermediated server and therefore decouples clients and servers.
- **IceBox** is the application server within the ICE middleware. This means IceBox is responsible for starting, stopping, and deployment of applications.
- **IceStorm** is a typical publish/subscribe service. Similar to other approaches, messages are distributed by their associated topics. Publishers publish to topics and subscribers subscribe to topics.
- **IcePatch2** is a software-update service. By requesting an IcePatch, service clients can update software versions of specific applications. Note however, that IcePatch is not a versioning system in the classical sense (e.g. SVN or CVS).
- **Glacier2** provides communication through firewalls, hence it is a firewall traversal service making use of SSL and public key mechanisms.

3.2.9 Transparent Inter-Process Communication (TIPC)

The Transparent Inter-Process Communication (TIPC) framework evolved from cluster computing projects at Ericsson. Comparable to Spread (see Section 3.2.12), TIPC is a group communication framework with its own message format. TIPC provides a layer between applications and the packet transport mechanism, e.g. ATM or Ethernet. Even so, TIPC provides a broad range of network topologies, both physical and logical. Furthermore, similar to Spread, TIPC provides the communication infrastructure to send and receive messages in a connection-oriented and connectionless manner. Furthermore, TIPC provides command line tools to monitor, query, and configure the communication infrastructure.

3.2.10 D-Bus

The D-Bus interprocess communication (IPC) framework, developed mainly by RedHat, is under the umbrella of the `freedesktop.org` project. D-Bus is targeting mainly two IPC issues in desktop environments:

- communication between applications in the same desktop session, and
- communication between the desktop session and the operating system.

From an architectural point of view, D-Bus is a message bus daemon acting as a server. Applications (clients) connect to that daemon by making use of the functionality (e.g. connecting to the daemon, sending messages,..) provided by the library `libdbus`. Furthermore, the daemon is responsible for message dispatching between applications (directed), and between the operating system, and potential applications (undirected). The latter typically happens when a device is plugged in.

3.2.11 Lightweight Communications and Marshalling (LCM)

The Lightweight Communications and Marshalling (LCM) project provides a library. Main purpose of the library is to simplify the message passing and marshaling between distributed applications. For message passing the library make use of the UDP protocol (especially UDP multicast), and therefore no guarantees about message ordering and delivery are given. Data marshalling and demarshalling is achieved by defining LCM types. Based on that type definition an associated tool generates sourcecode for data marshalling and demarshalling. Due to the fact that UDP is used as a underlying communication protocol no daemon or server for relaying the data is needed. From an architectural point of view LCM provides a publish/subscriber framework. To make the data receivable for subscribers it is necessary that the publisher provides a so called channel name, which is a string transmitted with each packet that identifies the contents to receivers. From a robotics point of view it is interesting to note that LCM was used by the MIT team during the DARPA Grand Challenge. According to the developers, the LCM library performed quite robust and scaled very well (during the DARPA Grand Challenge).

3.2.12 Spread Toolkit (Spread)

The Spread Toolkit, or shortly *Spread*[36], is a group communication framework developed by Amir and Stanton at John Hopkins University. Spread provides a set of services for group communication. Firstly, the abstraction of groups (a name representing a set of processes). Secondly, the communication infrastructure to send and receive messages between groups and group members. The group service provided by Spread aims at scientific computing environments, where scalability (several thousands of active groups) is crucial. To support this, Spread provides several features, such as the agreed ordering of messages to the group, and the recovery of processes and whole groups in case of failures.

3.3 Conclusions

As shown in this chapter, the communication middleware marketplace is large, with many different technologies, for many different domains and applications. Although it is very difficult to assess in an unbiased manner, a best effort was undertaken to objectively analyze and evaluate a number of well-known systems. The analysis allows the following conclusions to be drawn:

- In general, communication middleware technologies try to bridge the physical decoupling of senders and receivers (or clients and server(s)). From the software side, such decoupling is desired, because tight coupling leads to complexity, confusion, and suffering. All of the presented middleware technologies provide some means for decoupling (on different levels).
- The **Data Distribution Service** implements the decoupling of senders and receivers through message-orientation. However, **DDS** is still a very recent standard and very few open source implementation exists. Therefore, it might be too early to decide whether **DDS** is already an option for robotics or not.
- **JMS** and associated implementations are the de-facto standard for Java-based messaging. Regrettably, **JMS** is mainly limited to JAVA and therefore not an option for heterogeneous (in the sense of programming languages and platforms) environments, as mostly used in robotics.
- The group communication frameworks **Spread** and **TIPC** are mature frameworks for heterogeneous communication environments. However, the functionality provided by those environments are too limited for robotics, where features such as persistence are desireable.

- The most fully developed and mature communication middleware in that survey are **CORBA** and **ICE**. **ICE** and **CORBA** (with various implementations) showed to be a feasible middleware technologie in the robotics domain (as demonstrated in the Orca2 and Orocos framework). However, such full-fledged middleware solutions tend to be difficult to understand and to use.
- As demonstrated in ROS, the use of **XML-RPC** is feasible to develop lightweight communication environments and might be an option for robotics.

Chapter 4

Interface Technologies

4.1 Component Interface Models

Generally, a component can be considered in several different forms based on the context of their use and the component life cycle. Some notable such views[6] include:

- The *specification* form describes the behavior of a set of component objects and defines a unit of implementation. The behavior is defined as a set of interfaces. A realization of a specification is a component implementation.
- The *interface* form represents a definition of a set of behaviors/services that a component can offer or require.
- The *implementation* form is a realization of a component specification. It is an independently deployable unit of software. It needs not be implemented in the form of a single physical item (e.g. single file).
- The *deployed* form of a component is a copy of its implementation. Deployment takes place through registration of the component with the runtime environment. This is required to identify a particular copy of the same component.
- The *object* form of a component is an instance of the deployed component and is a runtime concept. An installed component may have multiple component objects (which require explicit identification) or a single one (which may be implicit).

In this section we will consider components from the interface form point of view. Our analysis so far shows that in addition to the component concepts and component-oriented programming attributes introduced so far, there are various design decisions that need to be made when developing with components, particularly in distributed computing environments.

Above all, the concept of *programming to interfaces* emphasizes that a component exposes only its interfaces as defined during design phase to the outside, i.e. to the developers eventualling using the component. It should not matter to a system developer how those interfaces/services were implemented. In this respect two approaches can be distinguished in software community, both of which have their own merits and drawbacks.

1. Generation of *skeleton/template implementations* from the given interface definitions. This is the approach which is commonly adopted in many well-known software standards such as CORBA and SOA. Here, a component developer has to define component interfaces at the very beginning. These interface definitions, usually written in special purpose language, are then parsed by a tool which generates empty templates for implementation code. Since

some of the well-known robotics software projects rely on above mentioned standards, they directly inherit this principle. Examples are OpenRTM[3] and Miro[37]. The drawback of this approach is that interface details have to be decided already at the beginning of the development, which is often a difficult task. After the skeletons have been generated there is usually no way to add new or modify existing functionality in the interface without modifying the implementation. The tool chains supporting this process lack the property of tracing back any changes from implementation level to interface level, a capability also known as *round tripping*. One way to escape this problem is not to allow a direct use of the automatically generated code but rather to inherit from it. This *specialized code* has all the functionality defined on the model/interface level, while being able to extend the base functionality of the generated code without any problem. For instance, SmartSoft adopts such an approach in its recent model-driven approach [38, 39].

2. Extraction of required interface definitions from the implementation. Unlike the previous approach, the decision which operations to expose in the interface is delayed until the time when component functionality is complete and ready to be deployed. A developer needs to indicate in his implementation code what interface he wants to expose, which is then extracted by a special parser tool. This is the approach taken, for instance, by Microsoft Robotics Developer Studio[40].

Additionally, *programming to interfaces* (when there is appropriate tool support) relieves a developer from the burden of developing glue code required in the presence of communication infrastructure, because this could be *standardized* through introduction of code templates for communication. Interface definition language-based code generation allows to overcome many difficulties related to writing code for distributed applications, but the developers still need to decide on the interfaces themselves. That is, they need to answer questions like:

- What data types should the arguments and return type of the interface operations have?
- What kind of call semantics should they support?
- Which operations should be made public in the interface?

From the developer point of view, to whom everything usually is just a method call, the approach often do not differ wrt. this definition. But it would make it much simpler for the user of a component interface not only to know about its return types and parameters, but also the *context* in which an operation should be invoked. Our survey shows that in most of the distributed software environments, regardless of their application domain, there is clear separation between the classes of interface methods, although this separation is often implicit and was not intended initially. In the following, we provide a generic description framework for the component internal models and their interface categorization approaches.

We begin the analysis by introducing the generic component, i.e. *meta-component* model, which specifies generic input and output signals as well as general internal dynamics of a component. As depicted in Figure 4.1, we assume that this meta-component has four kinds of I/O signals:

- Two for input/output or *data flow* (green arrows)
- Two for ingoing/outgoing commands or *execution flow* (blue and orange arrows).

This assumption draws from the control theory domain and based on the common denominator of I/O signals for a physical plant. External or incoming signals (analogous to disturbances in case of physical object) represent commands to operate/influence the component, whereas outgoing signals are issued by the component to interact with other components. These are the only aspects relevant when dealing with the *interface form* of the component (see Section 4.1). In case of the internal dynamical model of the component, it is often free form and could be represented in anything ranging from automata, Petri nets, state charts or decision tables. In most of the contemporary software systems, the internal dynamical model is represented by a finite state machine with a number of states and transitions among them. Additionally, from the structural point of view, a component can be seen as an aggregation of substructures, where these substructures could be representing communication, computation, configuration, or coordination aspects of a component model. At the same time, we would like to emphasize that a meta-component is a generic abstract entity and does not enforce many constraints.

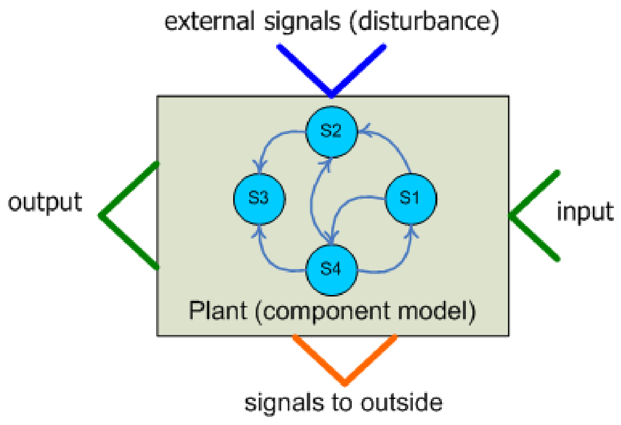


Figure 4.1: Meta-component model.

Interestingly, despite the generic component model defining two categories of interfaces, both in robotics and non-robotics domains researchers have managed to come up with many much more detailed component interface schemes. This refinement is based on different attributes of the interfaces, e.g. *context, timeliness, functionality* etc. Below, a list of the most commonly used *interface categorization schemes* in interface definitions is presented. There is no clear borderline between different schemes. In a particular component model, several schemes could be used in combination, often in the form of hierarchical interfaces. Such hierarchies could be not only conceptual but also implementational.

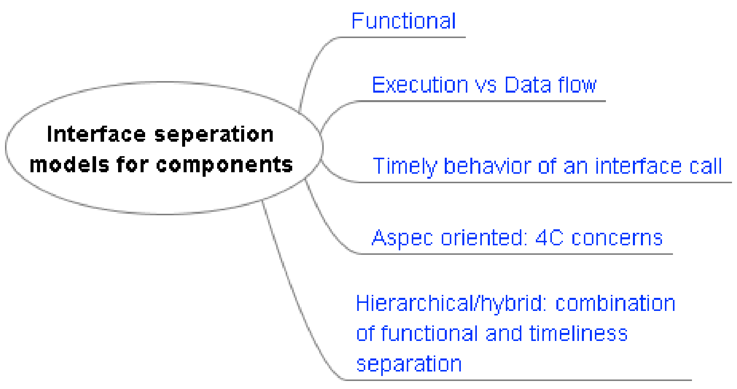

Interface seperation models for components
Functional
Execution vs Data flow
Timely behavior of an interface call
Aspec oriented: 4C concerns
Hierarchical/hybrid: combination of functional and timeliness separation

Figure 4.2: Interface separation schema.

- **Functional Scheme**: This type of categorization is based on the functionality provided by the interface, or by physical/virtual device the component represents. It is often in conceptual form and introduced to simplify the integration process for the user of a component.

Example: Let there be a component implementing image capturing functionality on a USB camera named *CameraUSBComp*. It performs simple image capturing from the device without any extra filtering/processing. Also, let there be another component *ImageEdgeFilterComp* which receives a raw image produced by *CameraUSBComp* and performs some edge extraction algorithm. Then, according to the functional separation scheme, the user of these components will only see, e.g. *CameraUSB/CameraUSBCapture* provided by the first component and *ImageEdgeFilter* by the second. So, when integrating the functionalities of these components into a system a developer needs not be aware of other technical aspects under these interfaces, such as their communication mode or timing requirements etc. Figure 4.3 depicts this situation. Often, this interface scheme serves as a group for more fine-grained public methods of the component.

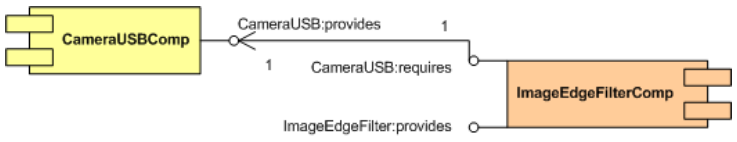


Figure 4.3: Topology representing functional interfaces of components.

- **Data and Execution Flow Scheme**: In this approach interfaces are categorized according to the type of information flow they carry. This could be either control commands or data. In most of the software systems where this scheme is adopted, the interface semantics is decoupled from the component semantics. That is, the former is related to communication aspect, whereas the latter is related to the computation aspect of the component. The decoupling is often reached through the concept of a *port*. This concept is not exclusive to this scheme, but can also be used with other schemes. It just turns to be often mentioned in the context of this approach. A *port* is virtual endpoint of the component through which it exchanges information with other components. An example to this approach could be a component with request/reply ports (which transmit execution flow information in synchronous mode), event ports (which transmit either execution or data flow information in asynchronous mode), and data ports (which transmit data flow information). Figure 4.4 below depicts a graphical representation of such a component.

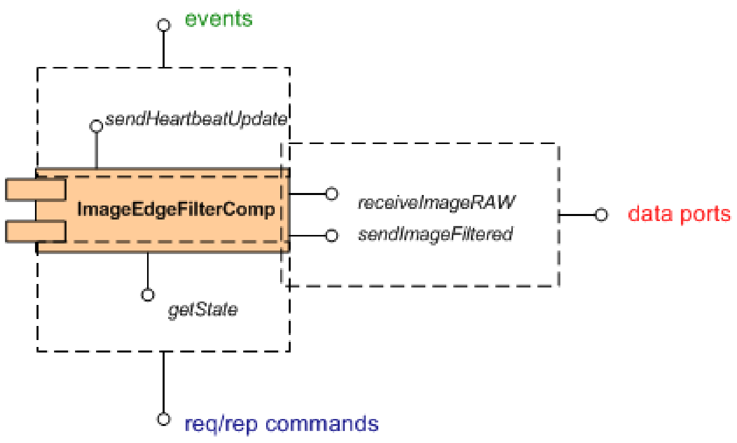


Figure 4.4: Interface categorization according to data and execution flows

- **Aspect-oriented: The 4C Concerns Scheme**: Another important design decision, not only in robotics but also in computer science, is separation of concerns. It defines a decomposition of a system into distinct features that overlap as little as possible. The separation of concerns can be considered orthogonal to an interface separation schema. It is often a system level process, but could equally be applied to component level primitives. Usually, four major concerns are identified in a software system [41]:

– *Computation* is the core of a component. This relates to the implementation of functionality that adds value to the component. This functionality typically requires read and write access to data from sources outside of this component, as well as some form of synchronization between the computational activities in multiple components.
– *Communication* is responsible for bringing data towards computational activities with the right quality of service, i.e., time, bandwidth, latency, accuracy, priority, etc.
– *Configuration* allows users of the computation and communication functionalities to influence the latter's behavior and performance, by setting properties, determining communication channels, providing hardware and software resources, and taking care of their appropriate allocation. It often relates to deployment aspects of the component.
– *Coordination* determines the system level behavior of all cooperating components in the system (the models of computation) from appropriate selection of the individual components' computation and communication behavior.

On the component interface level these aspects could appear as four classes of methods that the component requires or provides. This is very similar to the previous categorization scheme using data and execution flows. Figure 4.5 provides an example for how this might look like.

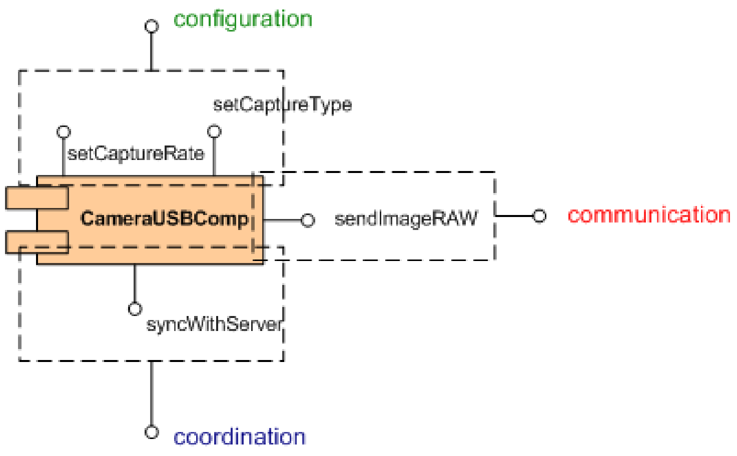


Figure 4.5: Component interface separation scheme according to 4C concerns.

- **Timing Property-Oriented Scheme**: This approach is very similar to a data and execution flow-oriented separation, that is, the interface classes have similar semantics (see Figure 4.4). The main difference is that the focus is not on the character of the flow transmitted but its timing characteristics. In other words, a distinction is made whether a method call is asynchronous vs synchronous, periodic vs aperiodic, etc. It is noteworthy that any synchronous communication is a special case of asynchronous communication with constraints on call return timing. Therefore, it can be assumed that one could design a component only with interfaces supporting asynchronous calls. But whether it makes really sense to do so is another question related to the requirements of transmission and the nature of data.

- **Hybrid Scheme**: This approach is a combination of functional and data and execution flow-oriented categorization. Interfaces are organized hierarchically (in previous cases, the hierarchy was mostly conceptual. In this case, hierarchy is used in the real application). On the top-most level, the interface has a functional description (i.e. it is used for grouping of operation) and can be used for composing a system based on the functionality provided by a particular component. Under the hood of functional interface are more fine-grained methods with data and execution flow semantics. For example, in the Player device server a particular device X may provide the functional interface `range2D` which can be used in system configuration file to connect with other components [25]. A system developer is not aware of transmission details concerning the timing attributes and semantics of the call, i.e. whether it is a command or data. But on the implementation level, `range2D` is matched with a method which carries particular transmission semantics. In Player, this semantics is defined in a separate file for all interfaces in Player. These message definitions are part of Player interface specification. For example, Listing 11 defines a message subtype for an interface providing/receiving `RangeData`, whereas Listing 12 defines a message for a command interface `CommandSetVelocity`. More details on the Player approach to component-oriented system development and interface technologies have been discussed in Section 2.1.

```
typedef  struct  player_ranger_data_range
{
  uint32_t  ranges_count;
  double *ranges;
} player_ranger_data_range_t;
```

Listing 11. Message specification for `RangeData` interface.

```
typedef  struct  player_position3d_cmd_vel
{
  player_pose3d_t  vel;
  uint8_t state;
} player_position3d_cmd_vel_t;
```

Listing 12. Message specification for `CommandSetVelocity` interface.

So far, we have been approaching components from a top-down perspective, i.e. through its interfaces to the external world. We saw that developers can use an interface specification given in the form of an IDL to generate component skeletons. Additionally, we analyzed how a set of interfaces attaching different semantics to the methods under that interface can be structured. This perspective is useful to build systems out of existing components, but it does not say anything concerning how the components along with their interfaces are implemented 2.1.1. Additionally, this question directly relates to how components internals are implemented, i.e. whether the component is in the form of a single class with methods, a collection of functions, a combination of classes with an execution thread, or any combination of these entities. Depending on the chosen method of component implementation, we can distinguish two main types of interfaces [26]:

- Direct or procedural interfaces as in traditional procedural or functional programming
- Indirect or object interfaces as in object-oriented programming

Often, these two approaches are unified under a single concept, by using static objects as a part of the component. Most component interfaces are of the object interface type (this is the case because most of the contemporary implementations rely on object-oriented programming languages). The main difference between direct and indirect interfaces is that the latter is often achieved through a mechanism known as dynamic method dispatch/lookup (as it is known in object-oriented programming). In this mechanism, a method invocation does not only involve the class which owns the invoked method but also other third-party classes of which neither the client nor the owner of the interface are aware of.

4.2 Hardware Device Interfaces and Abstractions

Hardware abstractions separate hardware-dependent issues from hardware-independent issues. A hardware abstraction hierarchy provides generic device models for classes of peripherals found in robotics systems, such as laser range finders, cameras, and motor controllers. The generic device models allow to write programs using a consistent API and reduce or minimize dependence on the underlying hardware.

4.2.1 The Need for Hardware Abstractions

- **Portability:** Hardware abstraction can minimize the application software dependencies on specific hardware. This allows to write more portable code, which can run on multiple hardware platforms.
- **Exchangeability:** Standardized hardware interfaces make hardware exchange easier.
- **Reusability:** Generic device classes or interfaces which can be reused by several devices avoid code replication. Due to the more frequent (re)use, the source code of generic classes is usually tested more thoroughly and more stable.
- **Maintainability:** Consistent, standardized hardware interfaces increase the maintainability of source code.

Benefits for Application Developers: Hardware abstraction can hide the peculiarities of vendor-specific device interfaces through well-defined, harmonized interfaces. The application developer does not have to worry about these peculiarities e.g. low-level communication to the hardware devices, and can use more convenient, standardized hardware APIs.

Benefits for Device Driver Developers: Each device interface defines a set of functions necessary to manipulate the particular class of device. When writing a new driver for a peripheral, these set of driver functions has to be implemented. As a result, the driver development task is predefined and well documented. In addition, it is possible to use existing hardware abstraction functions and applications to access the device, which saves software development effort.

4.2.2 Challenges for Hardware Abstraction

In the literature (see e.g. [42]), numerous challenges are described which have to be taken into account while designing interfaces (APIs) to hardware:

- **Interface Stability:** If a published generic interface is changed, a possibly very large number of applications depending on it must be changed. Therefore, generic interfaces have to be stable and mature. This can only be accomplished with experience and after thoroughly testing the interfaces on many heterogeneous robotic platforms. Generic interfaces should be neither the union nor the intersection (least common denominator) of the capabilities of the specific hardware devices. The solution often lies somewhere in between. Sometimes is is possible to stabilize the interface by using more complex data types.

- **Abstraction:** Abstraction should emphasize common functionalities and hide device-specific details not necessarily needed in the interface. Given an arbitrary hardware device and its often peculiar interface specification, identifying the essential functionality can be quite challenging.

- **Resource Sharing:** Many of the sensors and actuators of a robot are used by several functional components of a robot control architecture, and must be treated as a shared resources. In order to prevent access collisions and to ensure data integrity, device access may need to be protected by guards or monitors and by using reservation tokens to manage device access.

- **Runtime Efficiency:** Robotics application software developers seldomly can afford to trade performance for generality. Thus, anything providing more generality may incur only limited performance penalties with respect to a custom-built solution for interfacing the hardware device.

- **Hardware Architecture Abstraction:** Just generalizing hardware devices themselves is sometimes not enough to achieve interoperability across different robotics hardware platforms and exchangeability of hardware devices within different hardware architectures. Generalizations often implicitly assume a certain similarity wrt. the hardware architecture, e.g. how a device is physically interfaced with computational devices. In image acquisition, for instance, some systems connect high quality analog cameras via specific frame grabber boards (frame grabbers), while other systems use digital cameras directly connected via a serial bus like IEEE1395/FireWire or USB2.0. Multilevel hierarchical abstractions can provide a range of interfaces from most general to device-specific, and allow to overcome these problems at least partially.

- **Multifunctional Hardware Devices:** Some off-the-shelf robots consists of a base providing all functionality necessary for locomotion as well as a range of additional sensors. Often all the sensor and actuator hardware is controlled by a single PC or microcontroller board, which presents itself to the programmer as a single device. In these cases, it can be hard to provide clearly separated APIs for each sensor/sensor modality or each actuator.

- **Flexibility and Extendibility:** When aiming for general and flexible interfaces and trying to address as many use cases as possible, the abstraction hierarchy can get too complex, which makes it hard to extend and maintain. Sometimes, flexibility and generality need to be sacrifices for simplicity and improved maintainability.

- **Different Sensor Configuration:** Different robots may use different types and configurations of sensors for producing the same kind of or similar information. For instance, both a 3D laser scanner and a stereo camera can be used to produce 3D point clouds. By using an appropriate multi-level device abstraction hierarchy, these sensor devices could be made exchangeable.

- **Different Sensor Quality:** Even if two sensors produce the same kind of information, the quality of the information delivered (e.g. the noise level) can be very different. Both a laser scanner and a ring of sonars can produce range information, but the information from the laser scanner is usually much more accurate. The challenge is how to deal with such differences on the interface level, e.g. by providing additional quality-of-service (QoS) information.

- **Coordinate Transformations:** Sensors usually deliver information in terms of the coordinate systems associated with the device, while the modules processing such information usually need to transform this into another coordinate systems, e.g. a robot-centric coordinate system or a fixed world coordinate system. The coordinate transformations cannot be defined in isolation and require knowledge about the physical structure of the overall system, which defines the relationships between coordinate frames involved. Representation of coordinate frames should preferably be uniform in order to avoid difficult, inefficient and error-prone transformations when subsystems share such information.

- **Separation of Hardware Abstraction and Middleware:** If a hardware abstraction hierarchy is to be reused in another robotics software project, it hardware abstraction should be separated from communication middleware issues[31].
- **Distinction of Multiple Hardware Devices:** A robot can have multiple instances of the same hardware device, like two identical laser scanners, one on the front and another on the back. Usually, these two devices should be distinguishable by their operating systems port name (e.g. COM1 or ttyUSB1). However, the port identifier assigned by the operating system is not always deterministic and may depend on the order the devices are plugged in or registered when booting the systems. It is always possible to distinguish between multiple instances of the same hardware device, if each hardware device has a unique, vendor-assigned ID, such as serial number. However, this is not always the case, and especially low cost devices often lack this feature.
- **Device Categorization:** For some hardware devices it is is not easy to categorize them into a single device category. There are integrated hardware devices which fall in two or more categories. For example, an integrated pan-tilt-zoom camera or a robot platform with integrated motor controller and sonar sensors may be accessed through the same hardware interface. The problem only occurs when a device has to be categorized into a single category. There is no problem, if it is possible to categorize a device into multiple categories. This could be implemented e.g. by multiple inheritance. The pan-tilt zoom camera interface would inherit from the zoom camera interface and the pan-tilt interface.

4.3 Assessment of Hardware Abstraction Approaches

Whe now look a bit closer into the hardware abstractions of different robotics software frameworks, focusing on laser scanners as a particular class of hardware devices. We chose the laser scanner because almost all robotics software frameworks (except YARP) implement hardware device abstractions for it. In the next few subsections, we consider the hardware abstractions of the following framworks: Orca, ROS, Player, YARP, ARIA, MRPT, OPRoS and OpenRTM.

4.3.1 Orca

Hardware Abstraction Hierarchy Orca [43] uses mostly two levels in its hardware device hierarchies, only the range finder devices feature three levels. The first level classifies hardware devices in terms of functionality. In the case of the range finders, there is a further classification in terms of the type of range finder, e.g. laser range finders. The last level is then the actual hardware device driver which can either serve a group of hardware devices, such as a ring of sonars, or a single hardware device. The UML diagram in Figure 4.6 shows a section of the device classes hierarchy of the robotics software framework Orca.

Laser Scanner Example The interface to the laser scanners is defined by the LaserScanner2d interface definition, which inherits from the RangeScanner2d interface. These definition are done in the ICE slice language.

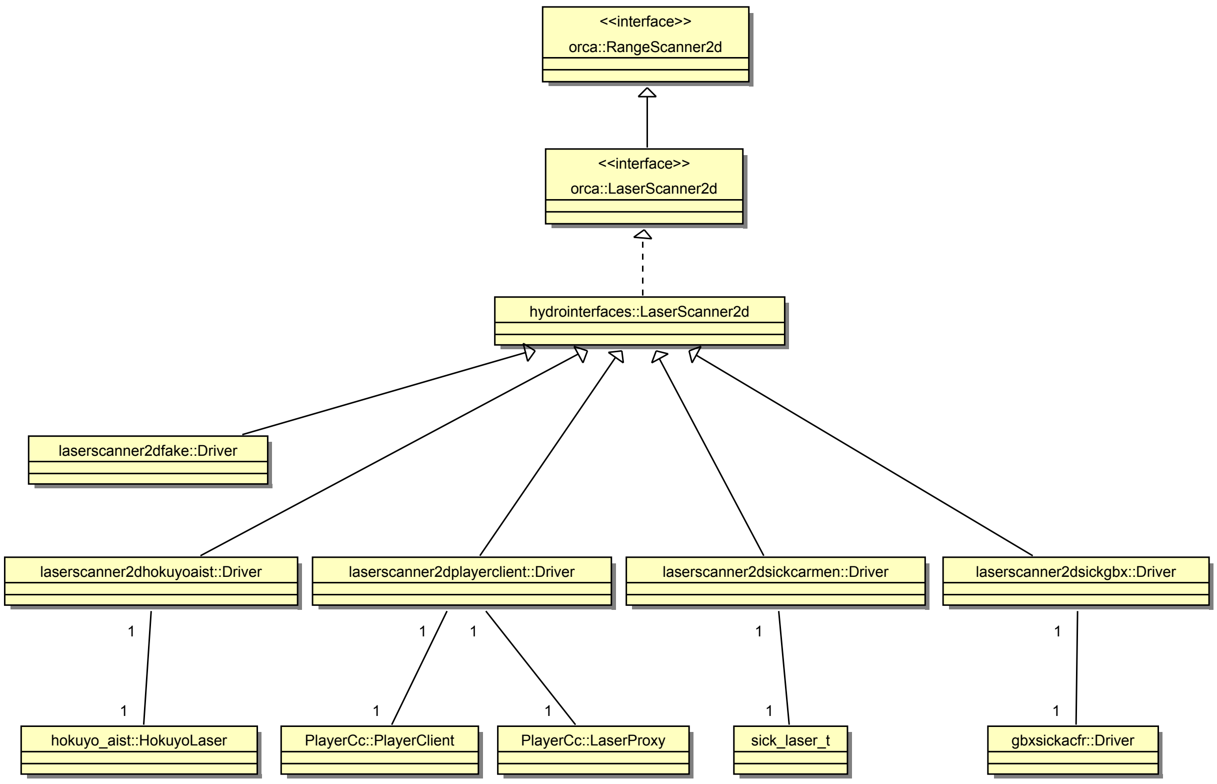

<<interface>>
orca::RangeScanner2d
<<interface>>
orca::LaserScanner2d
hydrointerfaces::LaserScanner2d
laserscanner2dfake::Driver
laserscanner2dhokuyoaist::Driver
laserscanner2dplayerclient::Driver
laserscanner2dsickcarmen::Driver
laserscanner2dsickgbx::Driver
hokuyo_aist::HokuyoLaser
PlayerCc::PlayerClient
PlayerCc::LaserProxy
sick_laser_t
gbxsickacfr::Driver
1
1

Figure 4.6: Orca range finder hardware hierarchy

In Orca, every hardware device is encapsulated in its own component. For instance, there is a component for a 2D laser scanner called Laser2d. This is a little bit inconsistent with the hardware hierarchy, because the component is not a generic range finder component. This is maybe the case because Orca does not support other range finders than laser scanners. The Laser2d component dynamically loads an implementation of a Hydro hardware interface hydrointerfaces::LaserScanner2d.

When configuring a device, a driver implementation has to be chosen. The following laser scanner implementations are available in Orca: Fake Driver, HokuyoAist, Carmen, Player or Gearbox. Generic parameters like minimum range [m], maximum range [m], field of view [rad], starting angle [rad] and the number of samples in a scan, are also set in the component. In the component one can also provide a position vector describing where the scanner is physically mounted with respect to the robot's local coordinate system. With this vector Orca is capable to hide the orientation of the laser scanner. Even if the scanner is mounted up-side-down, the clients can work with the scanner as it as usual (top-side-up), as long as the position vector has been set up correctly. The physical dimensions of the laser device can also be set in the component. All other configurations have to be done with the individual driver, like Carmen, Gearbox, etc. This means, that the configuration of devices is done on two different abstraction levels: on an abstract level, the parameters generic for all laser scanners (field of view, starting angle, number of samples in a scan), and on a lower level the parameters which have to be set individually for the specific device (port, baudrate).

4.3.2 ROS

Hardware Hierarchy In ROS[44], there are three packages in the `driver_common` stack which should be helpful for the development of a hardware device driver.

The `dynamic_reconfigure` package provides an infrastructure to reconfigure node parameters at runtime without restarting the nodes. The `driver_base` package contains a base class for sensors to provide a consistent state machine and interface. The `timestamp_tools` package contains classes to help timestamping hardware events.

The individual drivers use globally defined message descriptions as interfaces. Thereby, it is possible to exchange one type of laser scanner type with another. A replacement laser scanner just needs to implement the same messages. The hardware abstraction of ROS is very similar to the hardware abstraction of Player (see Section 4.3.3).

Laser Scanner Example The `HokuyoNode` class is directly inherited from the abstract `DriverNode` class. There is no generic range finder or laser scanner class. The `DriverNode` class provides the helper classes `NodeHandle`, `SelfTest`, `diagnostic_updater` and `Reconfigurator`. The specific self tests or diagnostics are added at runtime by the `HokuyoNode` class. Aside of these classes, the `HokuyoNode` class also implements the methods: `read_config()`, `check_reconfigure()`, `start()`, `publishScan()`, `stop()` and several test methods.

4.3.3 Player

Hardware Abstraction: In [45] it is stated that the main purpose of Player is the abstraction of hardware. Player defines a set of interfaces, which can be used to interact with the hardware devices. By using these messages, it is possible to write application programs which use the laser interface without knowing what laser scanner the robot actually uses. Programs which are written in such a manner are more portable to other robot platforms. There are three key concepts in Player:

- **Interface**: A specification of how to interact with a certain class of robotic sensor, actuator, or algorithm. The interface defines the syntax and semantics of all messages that can be exchanged. The interface of Player can only define TCP messages and cannot model something like a RPC.

- **Driver**: A piece of software, usually written in C++, which communicates with a robotic sensor, actuator, or algorithm, and translates its inputs and outputs to conform to one or more interfaces. By implementing globally defined interfaces the driver hides the specifics of a given entity.

- **Device**: If a driver is bound to an interface, this is called device. All messaging in Player occurs between devices via interfaces. The drivers, while doing most of the work, are never accessed directly.

Laser Scanner Example: The laser interface defines a format in which for instance a planar range sensor can return its range readings. The `sicklms200` driver communicates with a SICK LMS200 over a serial line and retrieves range data from it. The driver then translates the retrieved data to make it conform with the data structure defined in the interface. Other drivers support the laser interface as well, for instance the `urglaser` or the simulated laser device `stage`. Because they all use the same interface, it makes no difference to the application program which driver provides the range data. The drivers communicate directly by means of TCP sockets, which entangle the driver from the communication infrastructure. However, this reduces the portability of the Player drivers.

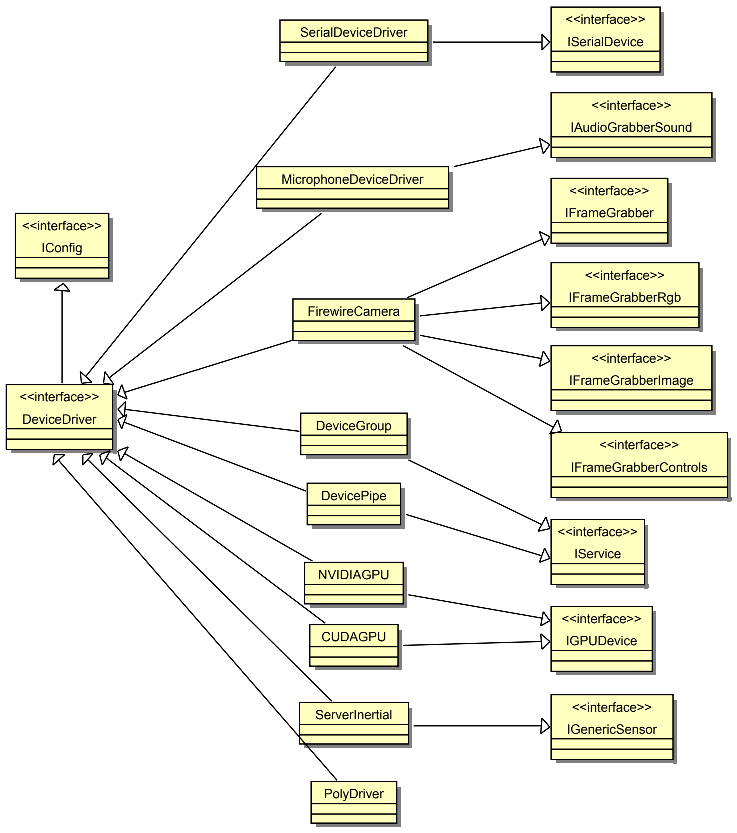


Figure 4.7: YARP Hardware Abstraction Hierarchy

4.3.4 YARP

Hardware Abstraction Hierarchy: In YARP[33], all device drivers inherit from the abstract class `DeviceDriver`, which itself inherits form the `IConfig` Class, which defines a configurable object.

```
yarp::dev::DeviceDriver : public yarp::os::IConfig
{
public:
  virtual ~DeviceDriver(){}
  virtual bool open(yarp::os::Searchable& config){ return true; }
  virtual bool close(){ return true; }
};
```

A subset of the abstraction hierarchy is illustrated in Figure 4.7. The hierarchy is in terms of interfaces. Classes with a body are not used. These interfaces shield the rest of your system from the driver specific code and make hardware replacement possible. The hierarchy provides harmonized and standard interfaces to the devices, which makes it possible to write portable application code not depending on specific devices.

The configuration process is separated out in YARP, in order to make it easy to control it via external command line switches or configuration files. Normally, YARP devices are started from the command line.

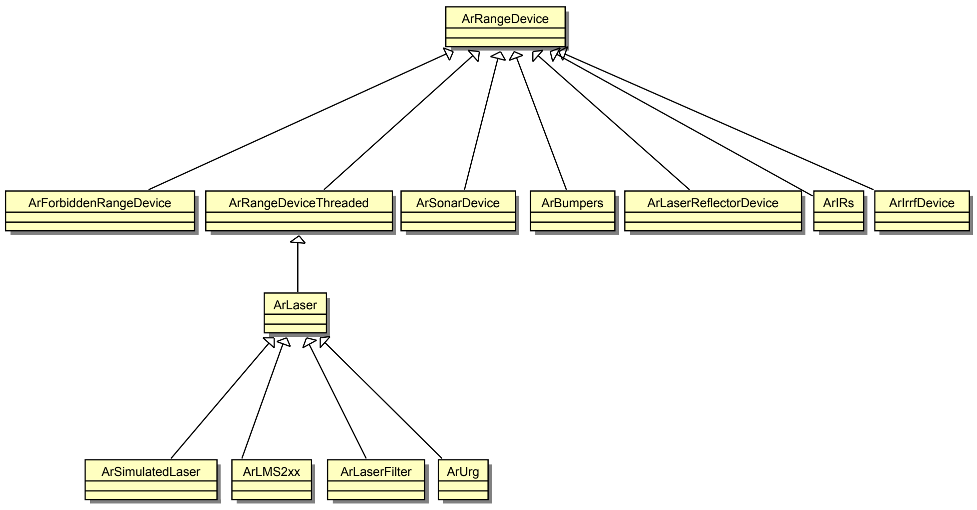


Figure 4.8: ARIA hardware hierarchy for range devices

4.3.5 ARIA

Hardware Hierarchy: Figure 4.8 shows the hardware hierarchy for the ARIA range devices. The `ArRangeDevice` class is a base class for all sensing devices which return range information. This class maintains two `ArRangeBuffer` objects: a current buffer for storing very recent readings, and a cumulative buffer for a longer history of readings. Subclasses are used for specific sensor implementations, like `ArSick` for SICK lasers and `ArSonarDevice` for the Pioneer sonar array. It can also be useful to treat "virtual" objects, for example forbidden areas specified by the user in a map, like range devices. Some of these subclasses may use a separate thread to update the range reading buffers. By just using the `ArRangeDevice` class in your application code, it is possible to exchange the hardware with all supported range devices. In theory, the application code is not only portable across all laser range finders, but also across all range devices. In practice, most of the application code makes implicit assumptions about the type of range device. Often, the application code assumes a specific quality of the sensor values, which may hold true only for some range devices. The `ArSick` class processes incoming data from a SICK LMS-200 laser range finding device in a background thread, and provides it through the standard `ArRangeDevice` API.

4.3.6 Mobile Robot Programming Toolkit (MRPT)

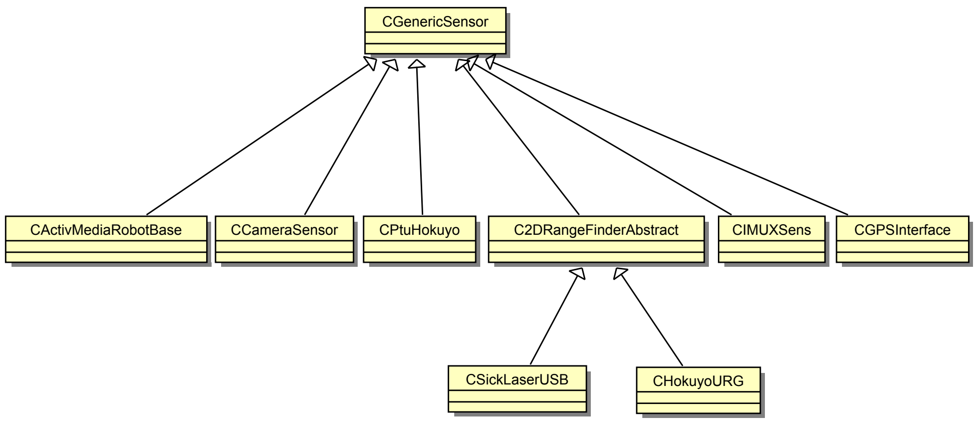


Figure 4.9: MRPT hardware hierarchy

Hardware Hierarchy: Figure 4.9 shows the hardware hierarchy of the Mobile Robot Programming Toolkit. The `CGenericSensor` class is a generic interface for a wide variety of sensors. The `C2DRangeFinderAbstract` is the base class for all 2D range finder. It hides all device specific detail which are not necessary for the rest of the system. The concrete hardware driver is selected by binding it to the `C2DRangeFinderAbstract` class. MRPT supports exclusion polygons, areas where points should be marked as invalid. Those areas are useful in cases where the scanner always detects part of the vehicle itself, and where these points should simply be ignored. Other hardware devices like actuators do not have a common base class.

4.3.7 OPRoS

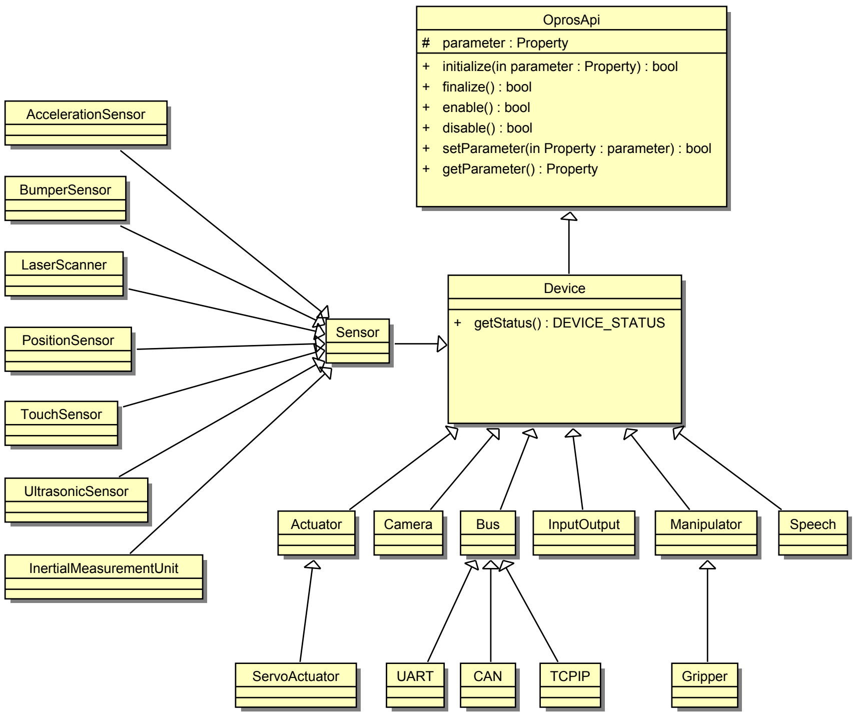


Figure 4.10: OPRoS hardware hierarchy

Hardware Hierarchy: The hardware hierarchy of OPRoS [46] is depicted in Figure 4.10. In this hierarchy, the sensor class and the camera class are side by side in the same level. We could not find any reasons why a `Camera` is not a `Sensor`. The same holds true for `Manipulator`, which is on the same level as `Actuator"`.

4.3.8 OpenRTM

Hardware Hierarchy: The OpenRTM project[47] does not define any hardware abstraction hierarchy. They just define interface guides for various devices and data types commonly found in robotics. OpenRTM defines interface guides for the following device types: **Actuator Array:** array of actuators, such as those found in limbs of humanoid robots, **AIO:** analog input/output, **Bumper:** bump sensor array, **Camera:** single camera, **DIO:** digital input/output, **GPS:** Global Positioning System device, **Gripper:** robotic gripper or hand, **IMU:** Inertial Measurement Unit, **Joystick:** joystick, **Limb:** robotic limb, **Multi-camera:** multi-camera, **PanTilt:** pan-tilt unit, **Ranger:** range-based sensor, such as infrared and sonar array, or laser scanner, and **RFID:** RFID reading device.

Laser Scanner Example: Some of the interface guides are used to define interfaces which are implemented by the Gearbox library[31]. The Gearbox library contains several drivers to hardware which are commonly used in robotics. The interfaces in the Gearbox library are not identical to the interface guides, they do extend the guides. Table 4.1 shows the interface guide for the range sensing device (left column) and the Sick laser scanner and Hokuyo laser scanner interfaces (center and right columns) which have been defined by using this guide.

Interface Guide:	**Sick Interface:**	**Hokuyo Interface:**
• Input ports: – None	• Input ports: – None	• Input ports: – None
• Output ports – ranges – intensities	• Output ports – ranges – intensities – errorState	• Output ports – ranges – intensities – errorState
• Service ports – GetGeometry – Power – EnableIntensities – GetConfig – SetConfig	• Service ports – None	• Service ports – Control
• Configuration options – minAngle – maxAngle – angularRes – minRange – maxRange – rangeRes – frequency	• Configuration options – StartAngle – NumSamples – BaudRate – MinRange – MaxRange – FOV – Port – DebugLevel	• Configuration options – StartAngle – EndAngle – StartStep – EndStep – ClusterCount – BaudRate – MotorSpeed – Power – HighSensitivity – PullMode – SendIntensityData – GetNewData – PortOptions

Table 4.1: OpenRTM range sensing device interfaces

Table 4.2 shows a comparison of the device interfaces and abstraction hierarchies of several major robotics software frameworks. The table provides information about the project website, the license under which it is available, and the supported programming languages and operating systems. The *Max. Depth* column contains the maximum depth of the abstraction hierarchy for a particular framework. The number of levels refers to the number of inheritance levels in the driver hierarchy; inheritance from generic classes or interfaces as in YARP or OPRoS is not counted. The "only Interfaces" column indicate if classes with a body are used in the

Framework	URL	License	Language	OS	Max. Depth	Interfaces Only	Message -Oriented	Coordinate Transform.
Orca	`orca-robotics.sourceforge.net`	LGPL[1] GPL	C++	Linux[2] Mac OS X	3 levels			√
ROS	`ros.org`	BSD	C++ Java Python	Linux Mac OS X	0 levels	√	√	√
Player	`playerstage.sourceforge.net`	LGPL	C++	Linux	0 levels	√	√	√
YARP	`eris.liralab.it/yarp/`	LGPL	C++	Mac OS X Linux Windows	2 levels	√		
ARIA	`robots.mobilerobots.com/wiki/ARIA`	GPL	C++	Linux Windows	4 levels			√
MRPT	`babel.isa.uma.es/mrpt`	LGPL	C++	Linux Windows	3 levels			
OPRoS	`opros.or.kr`	LGPL	C++	Linux Windows	3 levels			?
OpenRTM	`www.is.aist.go.jp/rt/OpenRTM-aist`	LGPL GPL	C++	Linux Windows Mac OS X	0 levels	√		√

Table 4.2: Hardware Abstraction Comparison Table

[a] A special clause in the Ice license allows them to distribute LGPL libraries linking to Ice (GPL)
[b] The interfaces, core libraries and utilities, and some components compile in Windows XP.

hierarchy or only interfaces without bodies. The "Message oriented" column indicate if the hardware abstraction interfaces are messages in the communication infrastructure. The "Coordinate Transformation" column indicate if the framework got an uniform representation of the coordinate transformations.

4.4 Conclusions

- Orca, YARP, ARIA, MRPT and OPRoS have multilevel abstraction hierarchies. They all classify hardware devices in a similar manner, by measured physical quantity (e.g. range). The robotics software frameworks derive the device drivers of devices measuring the same quantity form a generic driver class. This is reflected in the implementations by using the same return data type.
- Only YARP uses an abstraction hierarchy based on multiple inheritance. It is possible to classify a device in multiple categories.
- The robotics software frameworks use configuration files, command line input, or a configuration server to read the configuration of the devices. None of the frameworks introduces and uses separate configuration interfaces.
- ROS and Player got a similar concept to interface the devices. Both use predefined communication messages to unify the interface to a group of devices (e.g. laser range finder). By defining the interface with the communication message, they tightly couple the communication middleware with the device driver, there is no separation between communication and computation.
- All frameworks address the actual hardware device via the communication port name like 'COM3' or 'ttyUSB1'. None of them is using hardware device serial numbers to identify a device.

Chapter 5

Simulation and Emulation Technologies

5.1 Introduction

Simulation has a long-standing tradition in robotics as a useful tool for testing ideas on a virtual robot in a virtual setting before trying it on a real robot. However, when robots became an affordable commodity for research groups in the late 80s and early 90s, it became difficult to publish results that had not been obtained on real robots and in simulation only; as a consequence, simulation went almost out of fashion and was mainly used in specific sub-communities like swarm robotics and robot learning.

However, in the last decade, simulation in robotics is getting more attention again. One good reason for this is that the computational power of computers has been increasing significantly which makes it now possible to run computationally intensive algorithms on personal computers instead of special purpose hardware. Another reason is the increased effort of the game industry to create realistic virtual realities in computer games. The creation of virtual worlds requires a huge amount of processing power for graphical rendering and physics calculations. Thereby, the game industry developed software engines which, at least in principle, seem capable of providing high quality physic simulation and rendering software in the robotics domain. Since the goals of computer gaming and robotics simulations are quite similar — the creation of a realistic virtual counterpart of a real world use case —, robotics simulation environments can reuse these simulation and graphics engines and profit from the gain in processing power. The relationship between computer games and scientific research is more deeply analyzed in [48].

Below, we describe some motivations and benefits of using simulation in robotics; some descriptions also illustrate how simulators are applied in practice.

Speeding Up the Development Process: At the start of a new project or the development of a new product, the hardware often is not available. The time required for designing a robot, procuring off-the-shelf components, manufacturing custom-designed parts, and assembling the overall system can be significant. Rumors have it that this process can get so much delayed even in otherwise well-managed European research projects that projects are at risk to miss their original objectives because software development could start only after delivery of the hardware. Simulation can be of great help. Just as electronics design nowadays is able to develop graphics cards and motherboards long before the actual GPUs or CPUs become physically available, robotics needs the capability to develop software functionality without using the actual target hardware. Thereby, the availability of suitable simulators can speed up the development process enormously.

Producing Training Data for Offline Learning: Most learning techniques require a large amount of training data, which often is difficult and time-consuming to obtain using physical robots. Often, an appropriate simulation environment can be used to produce such data much faster and in almost arbitrary quantity. Also, in simulations it is often easier to generate such data with sufficient coverage and even distribution across the data space than using real robots. Even if the data produced by a simulation may be different from real-world data, especially with respect to the amount and distribution of noise, it is often sufficient for performing a first learning phase and generate functionality, which can be fine-tuned in a second learning phase using real-world data from a robot, however with much, much less training cycles to be performed on the physical system[49].

Speeding Up Interactive Learning: Some learning approaches make use of experience obtained from the interaction of the robot with its environment, e.g. reinforcement learning, genetic algorithms, and other evolutionary approaches. However, most of them need a large number of such interactions before they converge or even something useful could be learned. For example, Zagal presents an approach[50], where the control parameters for the four-legged robot AIBO have been learned based on simulation which took 600 learning iterations, each taking 20 sec in real time. Often, the number of iterations required is prohibitive for applying such approaches on real robots, and simulations, which execute virtual interactions at rates often exceeding 100 or 1000 times the interaction rate of a physical robot, offer the only way to apply these approaches and get acceptable learning effort.

Enabling Online Learning: When online learning approaches are used, it is sometimes very beneficial to apply learning experience acquired online to a number of similar virtual situations using simulation. Vice versa, alternative actions than the one that was selected and executed can be tried in simulation. Exploiting these opportunities often results in dramatic speed-up of the learning process.

Sharing Resources: Due to the high investment necessary for sophisticated robot hardware platforms, several researchers or developers may have to share the same hardware platform. This makes the robot platform a potential bottleneck and often leads to resource contention, especially when due dates or project reviews are approaching. The use of appropriate simulators can remedy this situation.

Permitting Distributed Development: Like in many large software projects, development of sophisticated software for service robotics is nowadays often performed by spatially distributed groups of researchers and developers. Examples include e.g. the recently completed German project DESIRE as well as many European projects, e.g. CogX or RoboCast. If the project has only a single platform available, simulation can serve here both for sharing resources and permitting distributed development. Sometimes, several platforms are used, but use different hardware components or different configurations. By using simulator models for each of these platforms and sharing them among the group, they can help to ensure that all developed software runs on all platforms in the consortium.

Compensating Resource Unavailability: The period an autonomous robots can be operated autonomously depends on its battery power and, depending on robot and battery size, ranges usually between less than an hour and a few hours. The time period needed to recharge the batteries often exceed the operation period by a factor of 2 to 4, during which the robot cannot be used for experiments involving navigation. Maintenance or calibration work that occassionally needs to be performed on the robot will further increase its unavailability. Simulators can help a lot to compensate for the unavailability of the real hardware platform.

Improving Safety and Security: Simulators can also help to improve safety and security. Especially when dealing with heavy or very fast robot equipment, programming errors could result in dangerous or damaging behavior with severe consequences. Test-driving such equipment first in a simulator can help to prevent such problems.

Increasing Thoroughness of Testing: Finally, simulators can be instrumental to increase the thoroughness of testing robotic equipment and applications, because they allow to safely assess a robot's behavior in unusual and extreme situations, which would otherwise be difficult to produce in real-world situations or which would incur potentially severe damages.

However, simulation also faces some tough challenges. Creating the models of the environments and the robots at a level of detail which allows simulation with believable physics and/or photo-realistic graphics is a very challenging task. Running these models in simulators can easily exceed the limits of even the most advanced computational hardware. Thus, simulation model builders must make difficult tradeoffs between model precision and runtime performance.

Another challenging problem is how to validate and debug models. There is little knowledge about systematic model validation in the community, as this is rarely in the curriculum of robotics courses. Complex simulation models can also be notoriously hard to debug, if model validation yields significant defects and aberrations.

Summarizing, there currently exists no general-purpose simulator that would qualify for all of the above tasks. In the following sections, different types of simulation environments and their properties will be described.

5.2 Simulators Used in Robotics

Since different research projects have different requirements with respect to the features and capabilities of a suitable simulator, a number of different simulation environments have been developed in robotics. Although many off them are somehow available at no or nominal cost, many systems are not maintained or further developed any more, sometimes already for many years. Such systems are difficult to use and adapt to up-to-date requirements concerning the supported platform models and graphics capabilities, and have little utility for new projects.

The differences in the requirements posed by particular projects are often related to the tradeoff to be chosen between simulator performance and model precision. Since it is not possible by principle to simulate a robot and its environment exactly as in the real world, different simulation approaches have evolved that focus on different problem aspects.

5.2.1 Low level Simulators

The simulators providing the highest precision with respect to physics are low-level simulators. They are usually used to simulate single devices, like motors or circuits, come with tools for the inspection of signals, and often do not support graphical visualizations of the robot and its environment.

A widely used commercial and very versatile environment is Matlab/Simulink by Mathworks[51]. It allows the graphical modeling of dynamical systems using block charts and the inspection and plotting of the generated output signals. Another low-level simulator is Spice [52], which is designed to simulate electric circuits.

Simulation environments like Simulink and Spice play only a small role in the software development of robots unless custom hardware is developed. It is more commonly used by hardware manufacturers producing actuators, sensor systems, or mechatronic components.

5.2.2 Algorithm Evaluation

Some simulators target a particular step in the robot development process, the selection of an algorithm or computational approach for a particular robot functionality. In order to make the right choice, we often need to compare and evaluate algorithms in specific settings. An example would be choosing a path planning approach for a mobile manipulator, built e.g. by combining one of several alternatives for a mobile base (differential, omnidirectional) and a particular manipulator. For such uses of a simulation environment, it should support the analytic evaluation of experiments by appropriate statistical tools.

GraspIt! is a simulation environment for robot grasp analysis [53]. It provides models for a manipulator (Puma 560), a mobile base (Nomadics XR4000), and a set of different robot hands. Different grasps can be evaluated and quality measures are given based on those described in Murry et al. [54]. For precise simulation of robot grasps, custom physics routines and collision detection routines were developed. The environment provides an interface to Matlab, a string-based TCP/IP interface, and is available in Linux and Windows versions. Custom models can be described in so-called *inventor models* which are quite similar to VRML.

OpenRave [55] is a successor of the RAVE project[56]. It is designed for robot manipulator path planning algorithm evaluation. External algorithms can interface the environment via Python, Octave, and Matlab, even over the network. The performance of the different algorithms can visualized in a 3D scene. In this environment, physics are only approximated, as no full-fledged physics engine is integrated. The purpose of this environment is to provide a common environment and robot model to objectively evaluate path planning algorithms and create statistical comparisons. The environment is not limited to manipulation and also supports the use of sensors like basic cameras and laser range finders. Furthermore, OpenRave provides an extensive list of model importers for COLLADA, a custom XML 3DS, and VRML. OpenRave is based on a plug-in based architecture which makes all modules in the system exchangeable.

5.2.3 Robot System Evaluation

The objective of the simulators described in this section is to evaluate complete robot systems. The simulated robot model can be controlled with the complete robot system and control architecture. Therefore, these systems provide a rich set of different sensors like ultrasonic sensors, infrared sensors, laser scanners, force sensors, or cameras to name only a few. Furthermore, different mobile robot bases and/or manipulators are often available as predefined models and can be used or adapted to build a custom robot model. Since the simulator and the robot control software should be interfaced in a manner that requires as little change to the robot code as possible, integrated solutions are provided, which combine a robot software framework and a simulation environment.

Player/Stage/Gazebo is the first environment that provided a combined approach of a robot software framework (Player) with an integrated 2D simulator (Stage) and later a 3D simulator (Gazebo) [57]. Since the 2D simulator is quite limited with respect to simulating sensors and manipulators, it is not further discussed here. The 3D simulator Gazebo builds upon the standard physics simulation engine ODE and can be natively interfaced with Player. Gazebo is now integrated into the ROS software framework and provides a binary interface called libgazebo. OpenGL visualization is provided via an Ogre library. Gazebo is running under Linux. New models can be created using C++ code, while the composition of models is done in an XML file. Gazebo is published under the GNU GPL license model.

Microsoft Robotics Developer Studio (MRDS) provides a web service-oriented software framework for robot development which is coupled with a graphical programming tool and a 3D simulation environment. The simulation environment is based on the commercial physic engine PhysX which allows hardware accelerated physic calculations. Simulations are visualized based on Microsofts XNA engine. The environment allows for the import of Collada models, but is commercially published.

USARSim [58] has been developed at CMU for the RoboCup Rescue competition and aims at the simulation of Urban Search and Rescue environments and robots. It is based on a commercial computer game engine name Unreal. The current version is ported from Unreal Engine 2, which was based on the Karma physics engine, to Unreal Engine 3, which uses the physics engine PhysX. USARsim requires a license of the game engine, but itself is licensed under GPL. USARSim is different from the aforementioned systems since it is not integrated in a robot software framework but it provides only interfaces to Player, MOAST[59] and Gamebots. USARSim is utilized for the RoboCup Rescue Virtual Robot Competition and IEEE Virtual Manufacturing Automation Competition.

Webots is a commercial simulation environment and a successor of the Khepera simulator[60]. It is a commercial environment based on the open source physics engine ODE and provides a world and robot editor that can create models graphically and store them in VRML. It has platform support for Linux, Windows and Mac OS X. Webots can be used to write robot programs and transfer them to a real robot, like the e-puck, the Nao, or Katana manipulators. C programs can be written to control simulated robots. There exist API's to C++, Matlab, Java and Python as well as the possibility to create TCP/IP interfaces.

Recent development such as Blender and Open Robot Simulator are still due for evaluation.

5.2.4 Learning Algorithms

An application area where simulation is very beneficial is learning, e.g. to obtain optimal control parameters for a walking robot. An essential requirement for simulators used for learning is excellent runtime performance since learning algorithms often need hundreds or thousands of iterations of the same experiment. Therefore, the simulator must be capable to run experiments faster than real-time. Nevertheless, these simulators often need to simulate physics so that the learned parameters are applicable in real world. Therefore, the environment may have be precisely simulated, e.g. when the robot has to learn how to climb stairs. The simulators discussed here often provide 3D visualization capability, but usually the visualization can be completely deactivated; for the learning outcome it is only important to record and archive the parameters providing the best performance.

A problem that simulation on this level faces eminently is the so-called reality gap[61]. It describes the problem that behaviors which are acquired in simulation may not behave exactly the same way or even completely fail if transfered to the real world. This directly relates to the complexity of the behavior or parameters that the robot shall learn.

UCHILSIM is a simulation environment developed for the four-legged league RoboCup competition at the University of Chile[50]. It uses the Open Dynamics Engine (ODE) as physics engine. Models are defined in a customized Virtual Reality Modeling language (VRML) format. Simulated robots can be controlled via an interface[62] based on the Open-R API for the AIBO robots. A unique feature of this simulator is the ability to learn simulation parameters of models [63]. The objective is to reduce the problems caused by the reality gap. By executing an experiment in the real world and in simulation simultaneously, the difference in the resulting behavior are used by evolutionary algorithm to adapt the simulation parameters. A limitation of this system is that it is very closely tied to the RoboCup application; it provides only a single robot model for Sony AIBO robot and a soccer field environment. Furthermore, this project received no updates since 2004.

YARS claims to be the fastest simulator for physical 3D simulation[64]. It targets the application of evoutionary learning algorithms and is also based on the ODE physics engine. The system can be interfaced by custom UDP clients, which exist for C++ and Java. Robot models can be described with the Robot Simulation Markup Language (RoSimL).

5.2.5 Learning Team Behaviors

The most abstract types of simulators are used for analyzing the behavior of multirobot teams or robot swarms. Since the precise simulation of the interaction of one or more complex robot systems with a reasonably sophisticated environment is computationally very challenging, these systems tend to use simplified robot and environment models. For swarm behavior, robots are often represented just as boxes and with simplified sensor models, which can be defined and configured without the support of sophisticated modeling tools and complex description languages and formats.

Mason is a simulator for multiagent simulation[65] and swarm robotics. It is one of only a few simulation environments developed in Java. To achieve high performance, an appropriate visualization level can be chosen: 3D, 2D, or no visualization at all. The models and the state of a simulation are completely decoupled, so that the simulation state can be stored or visualization mode can be changed at any time. The simulator contains collision detection routines, but no complete physics engine is integrated.

BREVE is a simulation environment for artificial life[66] and was not developed specifically for robotics. It can also be used for multiagent simulation and offers a custom language for agent description called *Steve*. The simulator offers a custom-developed physics engine and 3D visualization.

MuRoSimF is a special simulation environment for robot teams, but with flexible level of detail[67]. The simulation algorithms can be distributed over different CPUs. It has been developed for the RoboCup Humanoid soccer competition. The unique feature of this simulator is that it can be varied in precision of the algorithms for motion, sensing, collision detection, and manipulation. With this approach the appropriate level of detail and the resulting influence on simulation performance can be chosen based on the desired experiments. The simulator uses a custom-developed engine. Control software can exchange data with the simulated models via RS232 serial lines and/or TCP/IP ports.

5.3 Conclusions

The available simulators for robotics all have very specific target uses. There is no all-in-one solution which would be usable and suitable for any simulation-related task during the robot development process. Sometimes it may be necessary to use multiple simulation environments for a single project or robot development, each targeted towards solving a particular problem on hand, and requiring a particular tradeoff between model precision and performance to be chosen. The same holds true for the techniques used to build simulators. For example, for the physics engine of a simulator, open source engines like ODE or Bullet are available, but also commercial engines like PhysX, or custom-developed ones.

Another major issue is the integration of the simulation environment and the robot control software. Integrated solutions like Player or the MRDS exist, but there are no standardized interfaces which a simulator shall could support. Interfacing can be done via shared memory, custom API's, custom TCP/IP interfaces. Integration of simulators with commonly used software frameworks like Player and ROS are getting more wide-spread.

Furthermore, there are no standards available for modeling robots and environments, or even data exchange. Often, it is possible to store models in XML; standards developed elsewhere, like Collada and VRML, are becoming more frequently used but are not capable to persist models in a way that they can be exchanged between different simulation systems. The reason for that is that these standards have not been specifically designed for robot simulations and are missing important features like the description of sensor behaviors.

Many simulators evolve from particular research projects, and especially the RoboCup community provides many contributions in this area. A drawback of this situation is a lack of support and continuity in the development after some time, so that many good simulators become less and less usable or never result in a stable version.

Generally speaking, for a specific task an appropriate simulator has to be carefully chosen. Simulation can be a great help in the robot software development process, if the right choice of simulator is made with the right tradeoff between performance and model precision for the specific task.

Chapter 6

Conclusions

The amount of software technologies relevant for robotics is almost overwhelming. This makes a complete, unbiased assessment difficult. However, in the previous chapters we identified four major classes of software technologies relevant for robotics: component-based software technologies, communication middleware, interface technologies, and simulation and emulation technologies. With the BRICS objectives in mind, these technology classes were screened and analysed. The main objective of the survey was to analyze different approaches developed to tackle the problem of software development for robotics from different perspectives. Functional properties, differences, and similarities were identified, and most importantly implications for the future BRICS software development process have been derived. The outcome of this research will be used to base and justify new design and development decisions of future BRICS software.

6.1 Component Technologies

The analysis of component technologies can be summarized as follows:

- There is an increasing trend to use component-based software engineering in robotics. One explanation for this trend is the increasing number of collaborative robotics software projects, where several spatially distributed research groups work together. Additionally, from a systems developer point of view, it would be desirable to reuse readily developed software packages and combine them without much extra effort into a new application. This is exactly what component technology promises to deliver.

- From a programmer's point of view, there seems to be some convergence on component (meta-)models and the required associated tool chain support for developing component-based applications. In particular, this is observable for component modeling primitives. Most of the software frameworks reviewed adopt semantically very similar concepts, such as data and service ports for interfaces, stepwise component initialization, and running and clean-up for state machines.

- There is an increasing trend towards tool support for software management, e.g. for installation, clean-up, dependecy checking, etc. Future goals for tool chain providers should include design, development, and deployment of components for different application contexts.

- The missing links in software for robotics include interoperability and reuse between different software systems. Software can be viewed as a representation of developer's knowledge embedded in code. Every time someone else reimplements the same algorithm again in a different piece of code, he or she reinvents the wheel and wastes effort on the issues which have already been solved in some other software framework. In order to achieve higher efficiency and make more progress in focused areas, it is absolutely necessary to foster interoperability and reuse among existing software systems. The same situation was observed a decade ago in the operating systems domain. The introduction of interoperability between various UNIXes and Linux flavors led to the current situation, where almost any piece of code can be executed on any platform. This should also be one of the main goals in robotics software community.

6.2 Communication Middleware

All major robot software frameworks make use of communication middleware technology to achieve network and location transparency. However, there is a wide variety of robust and reliable communication middleware out there, all of which have been successfully applied either in robotics or in closely related domains. Currently, it is impossible to identify a single middleware solution which might eventually dominate the market and evolve as a de facto standard. Furthermore, opting for one particular solution seems currently very risky, because such a choice can lead to a premature technology lock-in. Within BRICS, this should be avoided. Facing this situation, we intent to pursue the following strategy:

- All functionality concerned with communication should be as clearly delineated and encapsulated from other aspects as much as possible. This will help to implement alternative solutions more quickly.
- BRICS will make an attempt to support more than one communication middleware. We hope to learn enough from the design and implementation of all the glue code to integrate two or more middlewares and to be able to design everything in a sufficiently generic manner such that switching to and integrating yet another middleware turns into an exercise consuming predictable time and effort.

6.3 Interface Technologies

After assessing the interfaces and hardware abstraction techniques of several robotics software frameworks, we can infer the following design goals for the object-oriented device interface level in the BRICS software architecture framework.

- The main objective for the BRICS OO device interface level should be to hide peculiarities of vendor-specific device interfaces by providing well-defined, harmonized object-oriented interfaces. These interfaces should be organized in a multi-level class abstraction hierarchy, which allows application programmers to develop with respect to generic, abstract interfaces.
- As some devices cannot be easily classified within a single inheritance hierarchy, the class abstraction hierarchy should support multiple inheritance.
- Sometimes it is not possible to uniquely distinguish hardware devices by OS port name only. Devices should be identifiable by a unique hardware serial number.
- In order to increase portability of OO device interface classes, a design approach adopting a separation of concerns strategy should be adopted. For example, functionalities concerning communication with or the configuration of the device should be separated from its core functionality.

- A harmonized set of data structures and ontological committments, e.g. for coordinate systems and their transformations, would further reduce complexity and increase maintainability.

We intent to apply these design goals to develop interface design guidelines for seamless integration of new hardware devices into BRICS.

6.4 Simulation and Emulation Technologies:

Simulation and emulation can be very useful tools in robotics software development. The use of simulators is widespread, but the selection of the type of simulator used and the precision of the models applied depend on the particular experiments and results the users have in mind. There is no single simulator that would be appropriate for all possible uses within the robot software development process. As a general requirement we can infer that

- simulators at various model precision levels are needed for environments and all animated and non-animated objects in the environment, and
- emulators, also at various model precision levels, for a wide variety of robotics platforms.

The use of the term emulator — analogous to the use of hardware emulation in chip design — is preferred and actively propagated by us, as we consider this an essential prerequisite for allowing the robot control software under development to be used without any change on both the emulator and the actual target hardware platform.

The use of simulation and emulation technology in robotics would benefit enormously from the availability of libraries of standardized, reusable environment models and robot platform models.

Bibliography

[1] "Orocos api reference." [online] http://www.orocos.org/stable/documentation/rtt/v1.8.x/api/html/index.html, 2007.

[2] P. Soetens, *The OROCOS Component Builder's Manual*, 1.8 ed., 2007.

[3] "Openrtm-aist documents." [online] http://openrtm.de/OpenRTM-aist/html-en/Documents.html, February 2010.

[4] H. Utz, S. Sablatnög, S. Enderle, and G. K. Kraetzschmar, "Miro – middleware for mobile robot applications," *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, vol. 18, pp. 493–497, August 2002.

[5] G. K. Kraetzschmar, "Best practice assessment of software engineering methods," BRICS Technical Report 002, Bonn-Rhein-Sieg University of Applied Sciences, Sankt Augustin, Germany, 2010.

[6] A. J. A. Wang and K. Qian, *Component-Oriented Programming.* Wiley-Interscience, 1 ed., 2005.

[7] P. Soetens and H. Bruyninckx, "Realtime hybrid task-based control for robots and machine tools," in *IEEE International Conference on Robotics and Automation*, pp. 260–265, 2005.

[8] N. Ando, T. SUEHIRO, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "Rt-component object model in rt-middleware- distributed component middleware for rt (robot technology)," in *IEEE International Symposium on Computational Intelligence in Robotics and Automation*, June 2005.

[9] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "Composite component framework for rt-middleware (robot technology middleware)," in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM2005)*, pp. 1330–1335, IEEE, 2005.

[10] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "Rt-middleware: Distributed component middleware for rt (robot technology)," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3555–3560, IEEE, August 2005.

[11] W.-K. Yoon, T. Suehiro, K. Kitagaki, T. Kotoku, and N. ANDO, "Robot skill components for rt-middleware," in *The 2nd International Conference on Ubiquitous Robots and Ambient Intelligence*, pp. 61–65, 2005.

[12] S. Fleury and M. Herrb, "Genom user's guide," 2003.

[13] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *International Journal of Robotics Research*, vol. 17, April 1998.

[14] B. Lussier, A. Lampe, R. Chatila, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell, “Fault tolerance in autonomous systems: How and how much?,” in *4th IARP - IEEE/RAS - EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, 2005.

[15] B. Lussier, R. Chatila, F. Ingrand, M. Killijian, and D. Powell, “On fault tolerance and robustness in autonomous systems,” in *3rd IARP - IEEE/RAS - EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments, (Manchester, UK)*, 7-9 September 2004.

[16] R. Alami, R. Chatila, F. Ingrand, and F. Py, “Dependability issues in a robot control architecture,” in *2nd IARP/IEEE-RAS Joint Workshop on Technical Challenge for Dependable Robots in Human Environments*, LAAS-CNRS, October 7- 8 2002.

[17] S. Fleury, M. Herrb, and R. Chatila, “Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture,” in *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 842–848, September 1997.

[18] K. Ohara, T. Suzuki, N. Ando, B. K. Kim, K. Ohba, and K. Tanie, “Distributed control of robot functions using rt middleware,” in *SICE-ICASE International Joint Conference*, pp. 2629–2632, 2006.

[19] Y. Tsuchiya, M. Mizukawa, T. Suehiro, N. Ando, H. Nakamoto, and A. Ikezoe, “Development of light-weight rt-component (lwrtc) on embedded processor-application to crawler control subsystem in the physical agent system,” in *SICE-ICASE International Joint Conference*, pp. 2618–2622, 2006.

[20] B. Gerkey, K. Stoey, and R. Vaughan, “Player robot server. technical report iris-00-392,” tech. rep., Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California, 2000.

[21] B. Gerkey, R. Vaughan, K. Stoey, A. Howard, G. Sukhtame, and M. Mataric, “Most valuable player: A robot device server for distributed control,” in *In Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1226–1231, 2001.

[22] B. Gerkey, R. Vaughan, and A. Howard, “The player/stage project: Tools for multi-robot and distributed sensor systems,” in *In Proc. of the International Conference on Advanced Robotics*, 2003.

[23] B. Gerkey, R. Vaughan, and A. Howard, “Player. version 1.5 user manual,” 2004.

[24] T. Collett, B. MacDonald, and B. Gerkey, “Player 2.0: Toward a practical robot programming framework,” in *In Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*, December 2005.

[25] “Player 2.0 interface specification reference.” [online] http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group–libplayercore.html, 2005.

[26] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Professional, 2 ed., 2002.

[27] “ROS developer documentation.” [online] http://www.ros.org/wiki/ROS/Concepts, January 2010.

[28] E. Kerami, *Web Services Essentials.* O'Reilly Media, 2002.

[29] S. Weerawarana and F. C. et.al, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More.* Prentice Hall PTR, 2005.

[30] "Ros roscpp client library api reference." [online] http://www.ros.org/wiki/roscpp/Overview/Initialization and Shutdown, January 2010.

[31] A. Makarenko, A. Brooks, and T. Kaupp, "On the benefits of making robotic software frameworks thin," in *IEEE International Conference on Intelligent Robots and Systems. Workshop on Evaluation of Middleware and Architectures.*, 2007.

[32] T. H. Collett, B. A. MacDonald, , and B. P. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*, 2005.

[33] G. Metta, P. Fitzpatrick, , and L. Natale, "Yarp: Yet another robot platform," *International Journal on Advanced Robotics Systems*, vol. 3, no. 1, pp. 43–48, 2006.

[34] D. Holz, J. Paulus, T. Breuer, G. Giorgana, M. Reckhaus, F. Hegger, C. Müller, Z. Jin, R. Hartanto, P. Ploeger, and G. Kraetzschmar, "The b-it-bots robocup@home 2009 team description paper." RoboCup-2009, Graz Austria, 2009.

[35] W. Emmerich, "Software engineering and middleware: a roadmap," in *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pp. 117–129, ACM, 2000.

[36] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton, "The spread toolkit: Architecture and performance," tech. rep., John Hopkins University, 2004.

[37] S. E. Hans Utz, Stefan Sablatnoeg and G. Kraetzschmar, "MIRO - middleware for mobile robot applications," in *IEEE Transactions on Robotics and Automation*, vol. 18, IEEE Press, August 2002.

[38] C. Schlegel, "Communication patterns as key towards component interoperability," in *In Davide Brugali, editor, Software Engineering for Experimental Robotics*, vol. 30, Springer, STAR series, 2007.

[39] C. Schlegel, "A component based approach for robotics software based on communication patterns: Crafting modular and interoperable systems," in *In Proc. IEEE International Conference on RObotics and Automation*, 2005.

[40] J. Jared, "Microsoft Robotics Studio: A Technical Introduction," *IEEE Robotics & Automation Magazine*, vol. 14, pp. 82–87, 2007.

[41] M. Radestock and S. Eisenbach, "Coordination in evolving systems," in *TreDS '96: Proceedings of the International Workshop on Trends in Distributed Systems*, (London, UK), pp. 162–176, Springer-Verlag, 1996.

[42] I. A. Nesnas, "The CLARAty project: Coping with hardware and software heterogeneity," *Springer Tracts in Advanced Robotics*, vol. 30, pp. 31–70, 2007. Jet Propulsion Laboratory, California Institute of Technology.

[43] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Orca: a component model and repository," *Software Engineering for Experimental Robotics. Springer Tracts in Advanced Robotics*, vol. 30, pp. 231–251, 2007.

[44] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *ICRA*, 2009.

[45] R. T. Vaughan, B. P. Gerkey, and A. Howard, "On device abstractions for portable, reusable robot code," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, 2003.

[46] B.-W. Choi and E.-C. Shin, "An architecture for OPRoS based software components and its applications," in *IEEE International Symposium on Assembly and Manufacturing*, 2009.

[47] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W. keun Yoon, "RT-Middleware: Distributed component middleware for RT (robot technology)," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2005)*, (Edmonton, Canada), pp. 3555–3560, 2005.

[48] M. Lewis and J. Jacobson, "Game Engines in Scientific Research," in *Communications of the ACM*, vol. 45, pp. 43–45, 2002.

[49] T. Gabel, R. Hafner, S. Lange, A. Merke, and M. Riedmiller, "Bridging the gap: Learning in the robocup simulation and midsize league," in *Proceedings of the 7th Portugese Conference on Automatic Control (Controlo 2006)*, September 2006.

[50] J. C. Zagal and J. R. del Solar, "UCHILSIM: A Dynamically and Visually Realistic Simulator for the RoboCup Four Legged League," in *RoboCup 2004: Robot Soccer World Cup VIII*, vol. 3276/2005, pp. 34–45, Springer, 2005.

[51] Mathworks, "Simulink." [Online] http://www.mathworks.de/, February 2010.

[52] P. W. Tuinenga, *Spice: A Guide to Circuit Simulation and Analysis Using PSpice*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1991.

[53] A. Miller and P. Allen, "GraspIt! A Versatile Simulator for Robotic Grasping," *Robotics Automation Magazine, IEEE*, vol. 11, pp. 110 – 122, Dec. 2004.

[54] R. M. Murray, S. S. S. , and L. Zexiang, *A Mathematical Introduction to Robotic Manipulation*. Boca Raton, FL, USA: CRC Press, Inc., 1994.

[55] R. Diankov and J. Kuffner, "OpenRAVE: A Planning Architecture for Autonomous Robotics," Tech. Rep. CMU-RI-TR-08-34, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, July 2008.

[56] K. Dixon, J. Dolan, W. Huang, C. Paredis, and P. Khosla, "RAVE: A Real and Virtual Environment for Multiple Mobile Robot Systems," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'99)*, vol. 3, pp. 1360–1367, October 1999.

[57] N. Koenig and A. Howard, "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator," in *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2149–2154, 2004.

[58] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "USARSim: a Robot Simulator for Research and Education," in *International Conference on Robotics and Automation*, pp. 1400 – 1405, 2007.

[59] C. Scrapper, S. Balakirsky, and E. Messina, "MOAST and USARSim: A Combined Cframework for the Development and Testing of Autonomous Systems," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 6230 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, June 2006.

[60] O. Michel, "Cyberbotics Ltd. Webots TM : Professional Mobile Robot Simulation," *Int. Journal of Advanced Robotic Systems*, vol. 1, pp. 39–42, 2004.

[61] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics," in *Advances in Artificial Life*, vol. 929/1995 of *Lecture Notes in Computer Science*, pp. 704–720, Springer, 1995.

[62] J. C. Zagal, I. Sarmiento, and J. R. del Solar, "An Application Interface for UCHILSIM and the Arrival of New Challenges," in *RoboCup 2005: Robot Soccer World Cup IX*, vol. 4020/2006, Springer, 2006.

[63] J. C. Zagal, J. R. del Solar, and P. Vallejos, "Back to Reality: Crossing the Reality Gap in Evolutionary Robotics," in *IAV 2004 the 5th IFAC Symposium on Intelligent Autonomous Vehicles*, 2004.

[64] K. Zahedi, A. von Twickel, and F. Pasemann, "YARS: A Physical 3D Simulator for Evolving Controllers for Real Robots," in *SIMPAR '08: Proceedings of the 1st International Conference on Simulation, Modeling, and Programming for Autonomous Robots* (S. Carpin, ed.), pp. 75 – 86, Springer, 2008.

[65] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "MASON: A Multiagent Simulation Environment," *Simulation*, vol. 81, no. 7, pp. 517–527, 2005.

[66] J. Klein, "BREVE: A 3D Environment for the Simulation of Decentralized Systems and Artificial Life," in *Proceedings of the Eighth International Conference on Artificial Life*, pp. 329 –334, 2003.

[67] M. Friedmann, K. Petersen, and O. von Styrk, "Simulation of Multi-Robot Teams with Flexible Level of Detail," in *SIMPAR '08: Proceedings of the 1st International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, (Berlin, Heidelberg), pp. 29 – 40, Springer-Verlag, 2008.