INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Development Environment for a Multi-Robot Team

Marco Alexandre Fagulha Barbosa

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Prof. Luís Rodrigues
Orientador:	Prof. Pedro U. Lima
Vogal Co-Orientador:	Prof. Arlindo Oliveira
Vogal:	Prof. Rodrigo Rodrigues

Novembro 2007

Abstract

Development of software for a multi-robot team is typically a daunting task. As more complex tasks are required from coordinated teams of robots, roboticists are faced with a huge increase in complexity of the needed underlying system, requiring an equal huge amount of effort to be built. This thesis describes the design and application of a development environment tailored to assist roboticists in easily overcoming recurrent problems in robotic applications and helping them concentrate their efforts in higher-level aspects of the robotic system. The development environment here described offers the roboticist a software middleware for abstracting from low-level details, a high-level software architecture for structuring the developed software and tools for runtime visualization and deployment of the system.

Keywords: robotics, development environment, middleware

Resumo

O desenvolvimento de *software* para equipas de vários robôs é, tipicamente, uma tarefa intimidatória dada a sua dimensão. À medida que são exigidas tarefas cada vez mais complexas a equipas de robôs, há também um enorme aumento na complexidade e no esforço necessário para construir todo o sistema subjacente à aplicação que se pretende. Esta tese descreve a concepção e aplicação de um ambiente de desenvolvimento com vista a ajudar a superar facilmente problemas recorrentes em aplicações robóticas. Deste modo, pretende-se que os investigadores concentrem os seus esforços em problemas de mais alto-nível. O ambiente de desenvolvimento aqui descrito oferece um *middleware* que abstrai os detalhes de baixo-nível, uma arquitectura de *software* de alto-nível para estruturar o *software* desenvolvido e ferramentas para a visualização em tempo de execução e o arranque do sistema.

Palavras-chave: robótica, ambiente de desenvolvimento, *middleware*

Acknowledgements

Aqui, nestas linhas, é pedido que o autor do imenso trabalho (supostamente) intelectual do texto que se segue agradeça às pessoas que, por alguma eventual razão, mereçam a delicadeza do acto. Gostaria apenas de salientar que não o faço por me ser pedido que o faça, muito menos por ser suposto que o faça. Faço-o, sim, porque estou sinceramente grato a todas as pessoas que contribuíram para que eu conseguisse chegar ao ponto de, imagine-se, completar um mestrado!

Gostaria de começar pelo mais importante: a minha Família. Tudo o que vocês são reflecte-se em mim. Obrigado Mommy, Daddy e Mana!!

Obrigado a todos os meus Amigos: há uma citação umas páginas á frente dedicada a todos vós. Estou certo que a irão compreender. :-)

Destaque para a grande EN Team! Nelson, João e Estilita: grandes tempos, heim? Obrigado por terem feito mais fácil esta passagem pelo "Inferno do Técnico" – que é quase tão difícil como o "Inferno da Luz" :-P

Aos SocRob'ianos! Todos os que participaram no projecto trouxeram algo de novo. Foi uma grande experiência: desafiante e uma fonte de imensas aprendizagens.

And last but definitely not least: gostaria de agradecer aos meus dois grandes Mentores que me proporcionaram o acesso a esta grande área que é a robótica: Prof. Ludgero Leote e Prof. Pedro Lima. Marcaram fortemente o meu percurso e as minhas escolhas nos últimos anos. Obrigado pelo empenho e pela motivação.

Marco

This work was partially supported by a grant from the project URUS Ubiquitous Networking Robotics in Urban Settings supported by the European Commission through contract # FP6-EU-IST-045062 and also by the Fundação para a Ciência e Tecnologia (ISR/IST pluriannual funding) through the POS Conhecimento Program that includes FEDER funds.

Table of Contents

List of Figures

List of Tables

Listings

List of Acronyms

ACE	Adaptive Communication Environment
ASCII	American Standard Code for Information Interchange
CORBA	Common Object Request Broker Architecture
DOM	Document Object Model
DSSA	Domain-Specific Software Architecture
FSM	Finite State Machine
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
ISR	Institute for Systems and Robotics
IST	Instituto Superior Técnico
MeRMaID	Multiple-Robot Middleware for Intelligent Decision-making
mlgen	MeRMaID Loader Generator
MSRS	Microsoft Robotics Studio
PC	Personal Computer
PDU	Protocol Data Unit
PNML	Petri Net Markup Language
REST	Representational State Transfer
SOAP	Service Oriented Architecture Protocol
SIF	Socrob InterFace
SOA	Service-Oriented Architecture
STL	Standard Template Library
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
XML	eXtensible Markup Language
YARP	Yet Another Robotic Platform

"Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius – and a lot of courage – to move in the opposite direction."

Albert Einstein

"I used to rock and roll all night and party ev-er-y day. Then it was every other day. Now I'm lucky if I can find half an hour a week in which to get funky"

Homer Simpson

1 Introduction

This work started to be developed in the SocRob project at the Institute for Systems and Robotics (ISR) – Lisboa. SocRob is a project on Cooperative Robotics and Multi-Agent Systems. The acronym of the project stands both for "Society of Robots" and "Soccer Robots" which has been the case study of the project since 1998. Due to the fact that soccer robotics is so rich in scientific challenges that are necessary to be overcome so that a team of robots may display adequate behavior, a huge amount of work has been done in many different areas. In fact, there has been some overlap with work done by other groups inside ISR, namely work related to vision and control. The absence of a unified framework and structuring of the developed solutions has led to the various groups implementing the same algorithms and functionality over and over again, without any sharing of code between the different groups. Even inside the SocRob project, the existence of two different robotic platforms made it desirable to have some middleware that would enable code sharing. The SocRob project was also in need of a tool that would allow the robots' behavior to be described in a more higher-level and with differing underlying formalisms.

Apart from the SocRob project that already had a defined software architecture prior to MeRMaID, there was no standard way of building software for multi-robot applications inside ISR. All the software needed for new projects was custom-built and difficult to integrate with other projects' code. With the beginning of the URUS project, which has a total of 11 partners involved, the need to have a structured way to do integration of all the software components became even more important. The URUS project (standing for Ubiquitous Networking Robotics in Urban Scenarios) aims at developing cooperation between humans and robots in urban areas, dealing with the coordination of several mobile robots, fixed sensors and the always unpredictable reactions of humans. For such a project, a big amount of software components have to be developed and integrated. From ISR's point-of-view, it would be desirable to develop these software components in such a way that they can be easily integrated and used in other projects.

Within this context and all of these needs, MeRMaID was created. MeRMaID and its associated tools aim at providing roboticists a more friendly environment in which to develop their research. It consists of a middleware layer – called MeRMaID::support – and a high-level software architecture. The work done in this thesis is concentrated on designing and building a new version of MeRMaID::support and additional tools – SocRob Interface (SIF) and MeRMaID Loader Generator (mlgen) – that are part of the development environment with which roboticists have to work.

SIF was tested in the SocRob project, while the new MeRMaID::support and mlgen started to be used in the URUS project.

This thesis will start by specifying the requirements of a development environment for multi-robot teams in Chapter 2. Chapter 3 focuses on the analysis of software similar to MeRMaID while Chapter 4 is devoted to the explanation of the newly developed MeRMaID::support, SIF and mlgen. Chapter 5 briefly shows third-party software that has been integrated with MeRMaID's development environment. An explanation of the applications to which the developed solution was applied and its results are present in Chapter 6. Finally, Chapter 7 contains the conclusions.

2 Development Environment Requirements for a Multi-Robot Team

Multi-robot systems, like any other reasonably complex system, benefit with the presence of a set of tools aimed at addressing issues that, although not being directly part of the problem in hand, need to be conveniently addressed. For instance, having a compiler for compiling the code is something that no project can live without, just like having a file editor for editing source code. Debugging tools are also extensively used in any kind of programming project. These three tools are normally the basis of so-called Integrated Development Environments (IDEs). Examples of those are Eclipse, KDevelop and Microsoft Visual Studio, just to name a few.

Besides IDEs, developers normally use several other software packages, known as software libraries, that implement needed functionalities. There are software libraries for many purposes and normally there are also several libraries to choose from in order to implement a specific functionality.

These IDEs and software libraries are built in order to be general-purpose and, although many of them are excellent tools, they are not sufficient to address the specific needs of multi-robot systems.

In the following sections, several requirements relevant for multi-robot systems are described.

2.1 Communication

Cooperative robotic applications can be seen, in general, as distributed systems that must be able to cooperate efficiently. This cooperation is achieved by means of the behaviors the robots are able to execute and the communication between them in order to ensure commitments, synchronization and, in general, other ingredients of teamwork. Communication between robots can be explicit (in which data is sent across a communication channel) or implicit (in which a robot perceives the intentions of his partners by observing their behavior).

Although there has been research on the topic of implicit communication, explicit communication is normally preferred due to the easily available communication systems that may be used (namely the IEEE 802.11 family of protocols) that provide high-bandwidth and low-latency in a wide range of situations. On top of these protocols, normal IP communication is possible, being up to the robotic system developer to choose which transport protocol to use (such as TCP or UDP) and, most importantly, to specify the details of the application layer protocol (following the five-level TCP/IP model [5]).

As there is no widely accepted standard for an inter-robot communication protocol, the robot developer is faced with the difficulty of needing to specify a new protocol for almost every project. Moreover, the developer also has to deal with integrating the defined communication mechanism within the other subsystems' software.

Therefore, we can define as requirements for a cooperative robotic application, to have clearly defined the following:

Low-level Protocol Stack The protocol stack should be clearly defined for the specific application, from the physical layer to the transport layer (adopting industry standard protocols is highly encouraged).

Application Layer Protocol The application layer protocol should be clearly defined, namely the PDUs and the logic behind their use.

Communication Framework A good framework for accessing the communication protocol should be given to the developer in order to use it correctly and efficiently. This framework should provide easy access to the underlying communication protocol and be as less intrusive as possible in its user's code.

2.2 Concurrency

Robotic systems are concurrent by nature. They are usually fairly complex systems that integrate lots of technology of different nature. To assemble a system of this kind, the developer is normally faced with the problem of concurrent access to shared resources. Ensuring reliable concurrent access while maintaining adequate performance may become a very challenging task as the system grows in complexity. Usually, locking mechanisms are employed to solve this, but then additional problems must be taken into account, such as deadlocks, livelocks and starvation (as illustrated by the dining philosophers example [7]). Also, with the latest developments of multi-core processors, one can expect these to be widely used in robotic applications. To take full benefits from multi-core processors a concurrent programming model must be followed.

The mechanism for concurrent access management also must be integrated with all the software developed for the system, so it should be as little invasive as possible (in the sense that it should not force the developer to take in consideration concurrency problems everywhere throughout the code). In short, concurrency issues bring the following requirements:

Concurrent Access Concurrent access to shared resources should be possible while maintaining a consistent state between accesses.

Concurrency Framework A simple framework for handling concurrency issues, such as access to shared resources and associated locking strategies, should be made available to the developer. This framework should be as little invasive in the rest of the code as possible.

2.3 Code Re-usability and Platform Independence

Robotic system developers often use similar approaches and/or algorithms for building robotic systems. More often than what is desirable, developers rewrite their code from scratch either due to poor software engineering practices or due to differences in the computing platform available for a specific application. This can be quite a burden for the developer since robotic systems are very diverse when it comes to hardware and operating-systems they run on. As such, this poses the following requirements:

Incentives for Code Re-usability The development environment should promote good software engineering practices by providing developers means to develop software components that are loosely coupled with each other.

Platform Independent Coding The developer should be able to build software that is independent of the underlying computing platform, so that the same code can be easily reused in another computing platform without change.

2.4 Deployment

Even if all the code developed is completely platform independent, it has to be executed in a particular machine running a particular operating system. This means that the code has to be somehow deployed in the target system. Depending on the target system this may be very easy or very complex. As an example of a very complex system for software deployment is Sony's AIBO robot that requires the developer to place binary files and a considerable number of configuration files in a fixed directory-structure. This illustrates the need for the following requirement:

Deployment Tool The user should have a tool that handles the generation of code and/or configuration files that are able to correctly deploy the code. By using this tool, the developer should be able to use the same deployment procedure for a wide number of robotic platforms.

2.5 High-Level Behavior Description Formalism

The biggest need for a roboticist is to have an easy way to describe the behavior of the robotic system and to quickly and easily change it. The representation of the behavior of the system should be in some kind of formal language. This poses the following requirements:

Formal Description of Robot Behavior The developer should be able to describe the desired behavior of the robotic system in some kind of formalism.

Robot Behavior Editor The description of the system's behavior should be editable with some kind of tool and that should be all that is necessary to change the system's behavior.

2.6 Runtime System Information

Multi-robot systems have to typically deal with large amounts of data coming from noisy sensors and have to act using actuators that also have a noisy output. Because of this, it is normally very difficult to understand exactly how the system is working and what it is really attempting to do. Even if all the sensors and actuators are simulated with some kind of software, it may still be very difficult to understand what the system is doing just by looking at how it behaves, especially when the system does not behave as expected. Therefore, while developing a multi-robot system the following is needed:

Runtime System Information Tool The developer should have a tool that enables him to inspect how the system is working by being able to read and visualize information relevant for assessing the state of the system.

3 Existing Solutions

3.1 Player/Stage

Player/Stage [9] is a collection of tools for interfacing sensors and actuators (with Player) and simulating them (with Stage). Player is a robot device server. Inside Player several so-called drivers interface with the physical hardware (both sensors and actuators). Client programs connect to Player to process data and send commands. Player defines a set of interfaces which the drivers follow so that client programs do not need to know which particular driver (and which particular hardware component) is sending data. This way, Player effectively acts as a hardware abstraction layer. Player provides a framework for client components to communicate with the server. Player does not, however, specify anything about client components and how should they be built or interact with each other.

Recently, Player has developed abstract drivers that instead of interfacing directly with the hardware use other drivers as sources for data and sinks for commands. These abstract drivers are intended to encapsulate useful algorithms so that they can be easily reused.

As for Stage, it serves as a 2D simulator for Player. Stage is able to simulate various robotic platforms, generating sensor data for virtual sensors, and commanding the virtual robot with commands received from Player's clients.

3.2 CLARAty

CLARAty [21] is a system architecture that is divided in two layers: functional layer and decision layer. The functional layer interfaces with the system hardware and its capabilities which are provided to the decision layer. The decision layer performs planning and control of the functional layer. It is further separated in a more high-level planning part and a lower-level executive part, although these are supposed to work as one. A big emphasis is put on describing the system at different levels of granularity and how that affects both layers' behavior.

CLARAty's code is entirely written in C++ and components are hierarchized with C++ class inheritance and aggregation. Linking or communication with code written in other languages is not supported. Components in the functional layer abstract from the underlying operating system by separating declaration and implementation. Class declaration is done using generic classes which state the class's interface, while specialized classes do the actual implementation of the class in a specific operating environment.

CLARAty also comes with an extensive framework, covering: motion control, manipulation, mobility and navigation, perception and vision, communication, resource management, system control and testing, verification and simulation. It was developed in NASA's Jet Propulsion Laboratory and certain parts of CLARAty have been recently released to public domain with an open-source license. More information can be found at http://claraty.jpl.nasa.gov/.

3.3 OROCOS

OROCOS [4], standing for Open Robot Control Software, is an attempt to develop a standard tool for robot control. OROCOS is composed of several libraries:

RealTime Toolkit the basis of component construction. The main programming primitive is the "TaskContext": an active Component which offers threadsafe and efficient ports for (lock-free) data exchange. It can react to events, process commands, or execute Finite State Machines in hard real-time. It can be configured online through a property interface (set/get values) and XML files. It also abstracts access to interfaces to common robotic hardware, such as encoders, AD/DA conversion, etc.

Orocos Component Library a collection of ready-to-use components that build upon the other libraries.

Kinematics Dynamics Library a framework for modeling and computing kinematic chains such as robots, biomechanical human models, computer-animated figures, machine tools, etc. It provides class libraries for geometrical objects (point, frame, line, ...), kinematic chains of various families (serial, humanoid, parallel, mobile, ...), and their motion specification and interpolation.

Bayesian Filtering Library a framework for inference in Dynamic Bayesian Networks, i.e., recursive information processing and estimation algorithms based on Bayes' rule, such as (Extended) Kalman Filters, Particle Filters (or Sequential Monte Carlo methods), etc.

OROCOS doesn't define any software architecture for how to compose the various components. That is left for the developer. OROCOS is based on CORBA for network communication and localization independence of components.

OROCOS also provides tools to aid testing components and relies on other Open Source Software to reduce development time and quickly build a working software middleware. Extensive documentation and some examples are available at http://www.orocos.org/.

3.4 MIRO

MIRO [20] is a middleware for robotic applications. It is strongly based on CORBA and is divided in three layers:

Miro Device Layer provides object-oriented interface abstractions for all sensory and actuatory facilities of a robot. This is the platform-dependent part of Miro.

Miro Service Layer provides active service abstractions for sensors and actuators via CORBA interface definition language (IDL) descriptions and implements these services as network transparent objects in a platform-independent manner. The programmer uses standard CORBA object protocols to interface to any device, either on a local or a remote robot. Also, event-based communication services based on the CORBA notification services are available.

Miro Class Framework provides a number of often-used functional modules for mobile robot control, like modules for mapping, self localization, behavior generation, path planning, logging, and visualization facilities.

MIRO doesn't define how components should be organized and leaves that task for the developer. MIRO is strongly based on CORBA and expects that the developer knows how CORBA works. Application code built by the developer will have to access directly the CORBA level. This is an abstraction level violation and may lead to a less confortable environment for the developer.

3.5 CARMEN

CARMEN [15] is an open-source control software. CARMEN's goals were to develop a consistent interface and a basic set of primitives for robotics research on a wide variety of commercial robot platforms. This would enable faster development with better sharing of developed code and algorithms between different platforms. CARMEN is divided in three-layers: the base layer is responsible for hardware interface and control. This layer also provides primitive control primitives such as following a straight line or doing simple rotations. Also done in this layer is information fusion from different sensors in order to improve estimates of, for example, robot odometry. There are components in the base layer capable of controlling a wide range of robots. The middle-layer is used for navigation control, including localization, dynamic object tracking, and motion planning. The top-layer is the application level which will use primitives made available by the middle-layer. For inter-component communication, CARMEN uses a framework called IPC which is described in [18].

3.6 YARP

YARP [13], standing for Yet Another Robot Platform, is a middleware concentrating mostly on the communication aspects between components. It provides communcation and thread synchronization primitives. A framework for signal processing is also available as well as a framework for device abstraction and access. YARP provides command-line tools for controlling and monitoring components and connections between components. The main abstraction entity in YARP is the `Port` object. `Ports` abstract communication in the way that the developer does not need to know in which underlying protocol the data is being sent. The `Ports`, in conjunction with the YARP name server, provide facilities for YARP components to run on several machines and make the component localization transparent to components who want to access it. Ports also provide advanced buffering techniques for data sent between components.

3.7 Microsoft Robotics Studio

Microsoft Robotics Studio (MSRS) [1] is Microsoft's attempt to establish itself as the main developer environment solution maker for robotics. This product is a result of integration of several other pre-existing products (such as the .NET platform and Microsoft Visual Studio) and some new ones developed specifically with robotic applications in mind. At the lowest level there is the Concurrency and Coordination Runtime that is responsible for low level scheduling and I/O. This low-level software follows a Representational State Transfer (REST) software architecture [8]. Communication is done using a newly defined protocol named Decentralized Software Services Protocol which is built upon SOAP. The communication protocol forces all software

components to respond to a very narrow set of pre-defined operations in an attempt that resources that code information in the same way can communicate with each other without additional effort.

Access to MSRS's low-level software is restricted to the use of C# and follows a series of patterns that the developer is forced to incorporate in his code. The developer has also to deal with details of the communication protocol in order to successfully develop a service, even though those details are not related to the service logic. A simple mechanism to regularly update a service's internal state is not provided, requiring extensive effort from the service developer in manipulating the underlying low-level framework.

A strong positive aspect of MSRS is the ability to formally define generic service contracts which are a means to specify standard interfaces between services. There is also a visual programming language and a visual simulation environment that promises to ease and make more efficient the development of robotic systems.

4 Developed Solution

To address the need of a suitable development environment and, more specifically, of a middleware targeted towards robotic applications, MeRMaID (Multiple-Robot Middleware for Intelligent Decision-making) [3] was created. MeRMaID is defined at two major levels: the first one is called MeRMaID::support which is the real middleware software and focuses its attention on the low-level issues such as service execution and communication. At the higher-level, MeRMaID defines a software architecture that gives the developer concepts and guidelines for building a sound multi-robot system such as robot behaviors.

In the first version of MeRMaID there was no hard separation between the support framework and the higher-level architecture. With the evolution of the code, including a full re-write and cleaner concepts, MeRMaID::support is now developed as a software package by its own. In this chapter we'll start by presenting MeRMaID::support and all the low-level mechanisms it offers to robotic system developers. We'll follow by describing the high-level MeRMaID architecture, focusing in the added value that having a clear high-level architecture offers, even though it may constrain the developers options. Finally, we will present two auxiliary tools that have also been developed: mlgen and SIF.

4.1 MeRMaID::support

MeRMaID::support is the base on which all of the rest of MeRMaID is based. MeRMaID::support consists of a collection of frameworks that the developer may use and which provide support for all the requirements defined in Chapter 2. The developer doesn't have to deal with all of the Frameworks that are part of MeRMaID::support, but they will all be presented here in order to explain in more detail the inner workings of MeRMaID::support.

4.1.1 ActiveObject Framework

This framework implements a simplified version of the Active Object pattern as described in [11]. The basic idea behind this pattern is to separate method invocation from method execution in order to simplify synchronized access to an object. In MeRMaID::support, an `ActiveObject` is a (C++) object which has its own thread of execution. Since C++ object methods are not first-class entities, developers may use, through inheritance, the `Task` object in order to add, in runtime, new "methods" for `ActiveObjects` to execute. `Task` objects are placeholder objects that contain a virtual method (named `run`) that the developer may overload with a specific implementation that will do whatever is intended to be done. `ActiveObjects` by themselves are only capable of receiving new `Tasks` and, later, executing them according to their defined schedule, since the `Task` also holds information about when it should be executed. What a `Task` does is entirely up to the framework's user. Since an `ActiveObject` only executes one `Task` at a time, the `Task` may access any resource belonging to that `ActiveObject` without concurrency concerns. All of the `ActiveObject`'s methods have a thread-safe implementation, so they can be called directly from any thread. These methods are the interface the framework's user has to control the `ActiveObjects`' behavior. `Tasks` should avoid doing actions that could result in blocking since that would effectively freeze the `ActiveObject`, as there is no mechanism for interrupting a `Task` in order to run another one.

4.1.2 Communication Framework

This is the framework where all communication resides. All the protocols for communication are implemented here and are used indirectly by other frameworks through the `CommunicationGateway` object. This allows for protocols to be implemented differently in different platforms and for the use of communication libraries to be completely transparent. Currently communication protocols have been implemented using YARP and that code is contained inside the Communication Framework. The communication protocol implemented over YARP is explained in detail in Section 4.1.7.

By using YARP several transport protocols are used, namely TCP and shared memory. UDP and multicast are also available but have not been used yet. The shared memory protocol is used whenever the two endpoints of the communication (the transmitter and the receiver) reside in the same machine so that a common shared memory zone may be created between both. Otherwise, TCP is used and communication is done through the normal TCP/IP protocol stack. The shared memory protocol is preferable to TCP because of its better performance as it consists of, at most, one memory copy, compared to the data flowing through the whole TCP/IP stack. This results in much lower latency and higher bandwidth compared to sending through TCP inside one machine.

The communication protocols are also responsible for serializing and deserializing data, handling differences in endianness (i.e. byte and bit ordering in memory) between systems. This is done by this framework in order to avoid unnecessary serialization/deserialization of data when the protocol used wouldn't need it. Therefore the communication protocols need to define standard representations of basic data-types that are the building blocks of all the data to be transmitted.

The basic data types that are serializable by YARP and used by MeRMaID::support are:

- integer;
- double;
- string;
- list.

All of the data to be transmitted has to be convertible to an arrangement of these basic data-types.

Even though this framework provides abstraction for communication it will not be used directly by MeRMaID::support users, but it is used internally by other MeRMaID::support frameworks, particularly the Service Framework.

4.1.3 Data Framework

Having defined communication protocols, it is also necessary to define the structure and meaning of the data that they communicate. The Data Framework addresses these issues and is divided in two parts: Data Structure and Data Semantics

4.1.3.1 Data Structure

As the name suggests, this part of the Data Framework addresses the issue of defining a common structure for data used with MeRMaID. Data structure can be described in an XML file and this description may be used,

for instance, to validate if data sent in Service Requests contains the correct data structure. The final format for this data structure description has still not been reached, due to attempts to make this format compatible with other formats used in other projects, namely the URUS project [17], and also to make it work more closely with the underlying YARP communication through which data has to be serialized and deserialized.

The description of a data element contains the following information:

data-name The name of the data element.

data-type The basic type of the data. Currently accepted values are: boolean, composite, float, integer and string.

data-value A textual representation of the value of the data.

For each data-type there must be a textual representation defined for that type. In Table 4.1 the textual formats for each of the basic data types are described.

Data Type	Textual Format
boolean	"true" | "false"
composite	XML descriptions of composing data items
float	[[(1-9)+ (0-9)* ,?] | [0,]] (0-9)*]
integer	[(1-9)+ (0-9)*]
string	any C-valid string

Table 4.1: Textual formats for basic data types

It should be noted that the boolean, float and integer are data-types with fixed-size while composite and string are not. Composite is the building block of more complex data structures and can be seen as a list-like structure in which other data types can be inserted (even other composites). In Listing 4.1 an example can be seen of a file describing a data structure that is a composite of a boolean, an integer, a float and a string.

Listing 4.1: data-structure-example.xml: Example of a simple data structure description file.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE data SYSTEM "data-structure.dtd">
<data>
  <data-name>bigComposite</data-name>
  <data-type>composite</data-type>
  <data-value>
    <data>
      <data-name>bool1</data-name>
      <data-type>boolean</data-type>
      <data-value>true</data-value>
    </data>
    <data>
      <data-name>int1</data-name>
      <data-type>integer</data-type>
      <data-value>42</data-value>
```

```
    </data>
    <data>
      <data-name>float1</data-name>
      <data-type>float</data-type>
      <data-value>3.14</data-value>
    </data>
    <data>
      <data-name>string1</data-name>
      <data-type>string</data-type>
      <data-value>Hello World!</data-value>
    </data>
  </data-value>
</data>
```

4.1.3.2 Data Semantics

The Data Semantics framework is an attempt to describe data not only by its structure, but to add to it semantic meaning. This framework can handle a tree of semantic nodes, being each node a concept. This way, ontology representations can be built and manipulated automatically in order to support more advanced functionality regarding manipulation and classification of data. This framework is currently implemented but not being used. It will prove its usefulness when automated data and service searching are implemented. The semantic tree may be read from an XML file. In listing 4.2 an example of a semantic tree is shown. With this example file, the Data Semantics framework is able to answer correctly to the question "are Mammals Plants?". Using this type of representation it is possible to build a whole ontology of data types and reason about it. For instance, and using this example file, if one would expect to validate some data and was expecting to receive Animal, the validation would be successful for any type of animal (like Mammals or Insects) but would return false for any kind of Plant (like Trees or Algae).

Listing 4.2: data-semantics-example.xml: Example of a simple semantic tree.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE semantic-tree SYSTEM "semantic-tree.dtd">
<semantic-tree>
  <root-node>
    <name>Living Beings</name>
    <semantic-node>
      <name>Animals</name>
      <semantic-node>
        <name>Mammals</name>
      </semantic-node>
      <semantic-node>
        <name>Insects</name>
      </semantic-node>
```

```
    </semantic-node>
    <semantic-node>
      <name>Plants</name>
      <semantic-node>
        <name>Trees</name>
      </semantic-node>
      <semantic-node>
        <name>Algae</name>
      </semantic-node>
    </semantic-node>
  </root-node>
</semantic-tree>
```

4.1.4 Error Handling Framework

The error handling framework consists only of a simple class named `Exception` that is intended to be the base class of all C++ Exceptions thrown by MeRMaID or user-developed components. This base class only holds a string that should be set to the reason for throwing the Exception.

4.1.5 Memory Management Framework

MeRMaID::support is implemented in C++ and therefore there is no garbage collection or automated memory management mechanism provided by the language. Since leaving memory management being done completely manually was not a viable option due to the amount of expected extra-work on coding and debugging the problems that this would bring, special template-based "smart pointers" were created. These pointer-like classes use a reference counting mechanism in order to track references to objects, deleting them when the reference count reaches 0. These pointers are also typed using C++'s template functionality. Therefore, inadvertently using one of these pointers to point to objects of a different type will result in a compilation error, preventing obscure bugs that could arise from using untyped pointers. These pointers are compatible with STL containers. These pointers should be used by value, even in containers, so no memory management is necessary even for the pointers themselves.

Two pointer types were created:

CountedPtr A simple reference count pointer. Each copy of the pointer increments the reference count and each destruction of the pointer decrements the reference count. When the reference count reaches 0 the object to which the pointer points is deleted. Throughout its life, the CountedPtr points always to the same object.

CowPtr This is a pointer that implements a Copy-On-Write mechanism. This pointer is used in order to avoid unnecessary copies of data that is going to be only read most of the times. If a write operation is attempted, a copy of the data is made and other users of the pointer may still access the original, unmodified, version of the object.

4.1.6 Service Framework

The Service Framework is the framework that developers using MeRMaID::support have to interact with most of the time. This framework is what enables MeRMaID to be a Service-Oriented Architecture (SOA). The design of this framework can be considered as an implementation of the abstract framework described by the "OASIS Reference Model for Service Oriented Architecture 1.0" [16].

In MeRMaID a Service is viewed in the same way as in the OASIS SOA Reference Model, that is: *"A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description"*. Therefore, a Service will hold the implementation of a certain functionality that is relevant for the system and whose access mechanisms are clearly defined. The Service Description holds all the information that is necessary to access and effectively use a determined Service. In MeRMaID a Service Description consists of an XML file which holds the following information:

name The name by which the Service should be known.

comment An explanation of the functionality that this Service offers.

service-interface A description of all the interfaces available for this Service.

data-feed A description of all the Data Feeds provided by the Service.

update-frequency The frequency at which the Service's update method should be called.

For each Service Interface the following information is written in the Service Description:

name The name of the interface.

comment An explanation of what is possible to achieve by using this interface.

in-data A description of the expected input data for this interface which is compatible to the Data Structure XML representation format presented earlier.

out-data A description of the expected output data for this interface which is compatible to the Data Structure XML representation format presented earlier.

Likewise, for each Data Feed the following information is present in the Service Description:

name The name of the Data Feed.

data A description of the expected data exported by this Data Feed which is compatible to the Data Structure XML representation format presented earlier.

Access to Service Interfaces is done through Service Requests. A Service Request is what travels between Services and is processed in the target Service after arriving to the Service Interface. Therefore, the Service Request should contain the input data for the target Service Interface as well as information about the target Service Interface in order for it to be delivered.

In Figure 4.1 the Service Request conceptual mechanism is illustrated. Service A sends a Service Request to Service B. The Service Request carries information about which Service Interface it is targeted to. After the service receives the request it should "process" it doing whatever is promised in the Service Description. In response to each and every Service Request, a Service Reply should be sent back to the request sender (in the example, that is Service A).

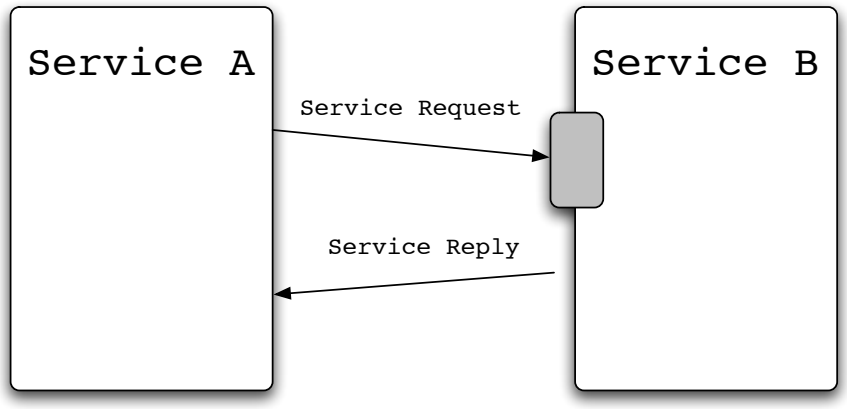

Figure 4.1: Service Request conceptual diagram.

Sending a Service Request to a Service Interface triggers some kind of action inside the Service. Although the mechanisms are the same, we may distinguish, for illustrative purposes, between three different kinds of actions triggered by the reception of Service Requests that typically occur:

stateless data processing The data received on the input is somehow processed and sent to the output. Every time the same data is sent, the same output is produced, thus the output doesn't depend on the internal state of the Service. For instance, imagine a Calculator Service that is able to sum two integers. The output of the sum of two given integers is always the same regardless of the internal state of the Calculator Service.

state dependant data processing The data received on the input is processed taking into account the state of the Service. Using the Calculator Service example, imagine that it implements an internal accumulator. Every time a Service Request for the accumulator arrives with an integer, it is added to the accumulator and the new value of the accumulator is returned.

state changing The state of the service is changed. The request may contain data or not and there is no output data.

It should be noted that the behavior of a Service may change through time, in which case time is considered as being part of the variables that define the state of the Service. Due to the fact that in robotic applications it is normal to have the need for updating the Service's internal state in regular time intervals (like reading sensors in regular time intervals or time-dependent algorithms that have to be run in a fixed time period), MeRMaID::support provides the ability for Services to set a regular interval in which a method called "`update`" is called. This information is present in the "update-frequency" item of the Service Description.

The other mechanism that enables Services to interact with each other is the Data Feed. Data Feeds are channels for Services to export data. The rate and the timing at which the data is exported is up to the Service that is producing the data. Other Services may connect to existing Data Feeds and use the data received for any purpose of its interest. In figure 4.2, Service A has two data feeds. In one of them, two Services are connected, while in the other only one Service is connected.

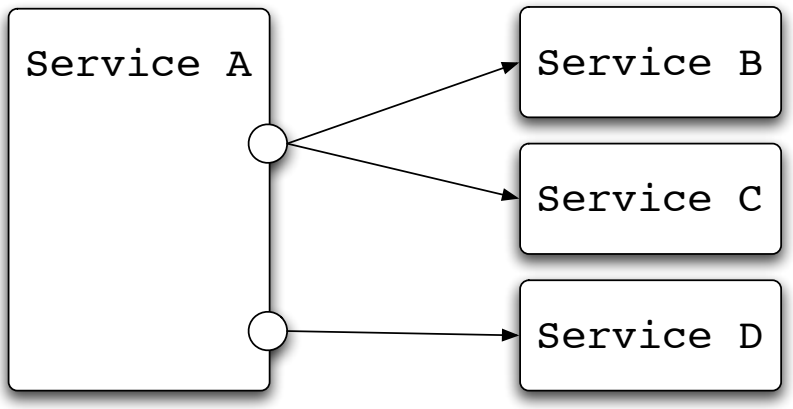

Figure 4.2: Data Feed conceptual diagram.

4.1.7 Communication Protocol

As described in Section 3.6, YARP offers a communication abstraction based on the `Port` concept. `Ports` are connected to each other in order to establish a data link. Every `Port` has a name that is registered in a name server. The name server is responsible for translating `Port` names to the `Ports`' network location. This information is then used by YARP to establish connections between `Ports`.

The YARP-based communication protocol here defined is implemented inside the Communication Framework described in 4.1.2 and implements directly the mechanisms defined for interaction between Services in the previous section.

This section will follow with the definition of the protocol for both of the Service interaction mechanisms (Service Request and Data Feed) and will end with a formal definition of the naming convention for the YARP `Ports`.

4.1.7.1 Service Request Mechanism implemented in YARP

To implement the Service Request interaction mechanism, each Service has to create YARP `Ports` following the port naming convention and send data in the formats here defined.

The Service Request interaction mechanism between two Services involves a total of four YARP ports. A Service willing to receive Service Requests has to open an input port where it will receive them. To send the Service Reply, an output port has to be also created. A Service wishing to send Service Requests has to send them through a port (called the request port) and receive the Service Replies on another one (called the reply port). An illustration of the arrangement of these ports can be seen in figure 4.3;

The sequence of actions to be taken by a Service sending a Service Request is:

1. Create the Request Port.
2. Create the Reply Port.
3. Build the YARP Bottle containing the Service Request.
4. Connect the Request Port to the other Service's Input Port.
5. Write the YARP Bottle to the Request Port.

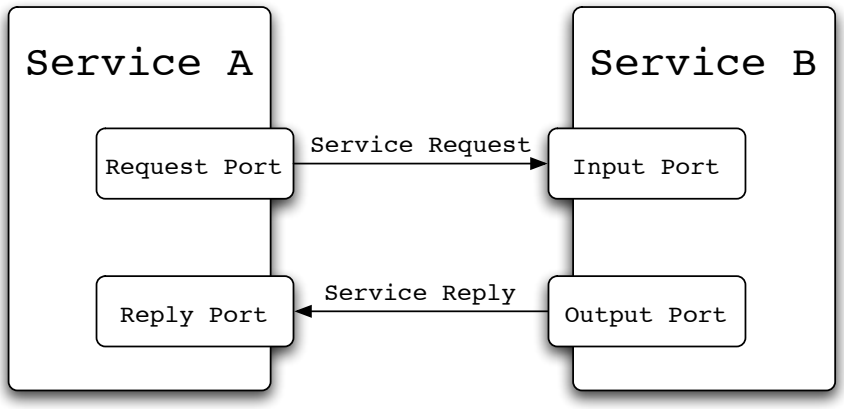

Figure 4.3: YARP ports for the Service Request interaction mechanism.

6. Read the Reply Port to receive the Service Reply.

Likewise, for the Service that is receiving Service Requests it should do the following sequence of actions:

1. Create the Input Port.
2. Read the Input Port to receive the Service Request.
3. Process the Service Request.
4. Create the Output Port.
5. Build the YARP Bottle containing the Service Reply.
6. Connect the Output Port to the requesting Service's Reply Port.
7. Write the YARP Bottle to the Output Port.

Service Requests are sent in YARP `Bottles`. YARP `Bottles` are special containers provided by YARP that behave as simple arrays of data and that have a well-defined binary representation, suitable for network transmission. All of the data going through YARP `Ports` is, by option, inside a YARP `Bottle`. This makes it possible to use tools provided by YARP to inspect the data being received and sent by a `Port` without interfering with the normal execution of the system. The `Bottle` representing a Service Request should have the following contents in the order and with the data type here described:

1. string: Entity name - the name of the entity in which the Service is running
2. string: Client Service name - the name of the Service which is sending the request
3. string: Service Interface Name - the Service Interface to which the Service Request is targeted
4. int: Request ID - an ID number of the Service Request being send. This number should be unique for all of the requests sent by a client Service
5. Bottle: Service Request Data - the data sent in the Service Request. The specific format of the data sent inside this Bottle is up to the target Service to define.

Service Replies, like Service Requests, are also sent in YARP `Bottles`. The order and data types of the elements sent in them, are:

1. string: Client Entity name - the name of the entity in which the Client Service is running
2. string: Client Service name - the same that came in the request that originated this reply

3. int : Request ID - the same that came in the request that originated this reply
4. string : Service Execution Status - it contains the string 'OK' if service ran successfully, otherwise it contains a string with the description of the error
5. Bottle: Service Reply Data - the data received with the Service Reply. The specific format of the data sent inside this Bottle is up to the target Service to define.

If the service execution status is not 'OK' the reply data Bottle is either empty or has the same structure as the Service Reply Data (it has to be always a valid YARP Bottle).

4.1.7.2 Data Feed Mechanism implemented in YARP

To implement the Data Feed interaction mechanism the Service producing the data has to create an Output Port where the data is exported and the Services that want to receive the data have to create an Input Port where they receive the data. Services that want to receive the data from a Data Feed are also responsible for connecting the Output Port to their Input Port. Data sent through a Data Feed is always encapsulated inside a YARP Bottle. In figure 4.4 an example of the layout of these Ports can be seen.

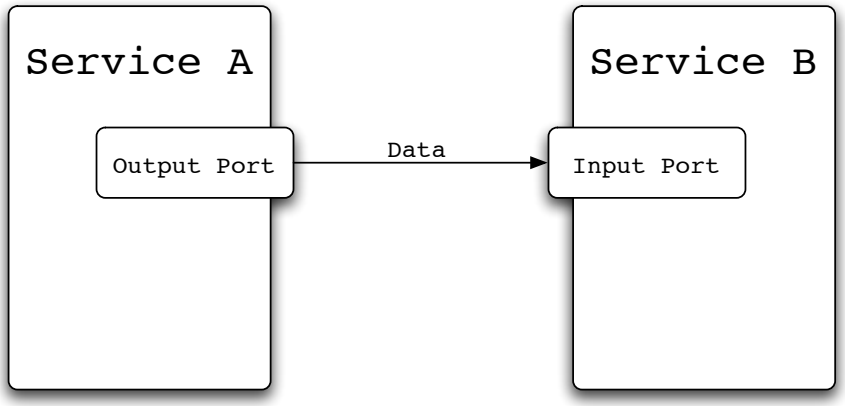

Figure 4.4: YARP Ports for the Data Feed interaction mechanism.

The sequence of actions to be taken by a Service that wants to export data via a Data Feed is:

1. Create the Output Port.
2. Build the Bottle containing the data to be sent.
3. Write the Bottle to the Output Port.

Likewise, the sequence of actions to be taken by a Service that wants to receive data from a Data Feed is:

1. Create the Input Port
2. Connect the other Service's Output Port to its Input Port
3. Read data from the Port

The Data Feed's data is sent encapsulated inside a YARP Bottle. The contents of the YARP Bottle should be defined by the Service that is exporting data.

4.1.7.3 YARP Port Naming Convention

The YARP Ports described in the previous sections should be named using the naming convention described here in BNF notation:

Listing 4.3: YARP `Port` naming convention for th Service Request and Data Feed mechanisms.

```
<port-name> ::= "/" <entity-name> "/" <service-name> "/"
    <interaction-specific-suffix>

<interaction-specific-suffix> ::= "service-request/␣"<service-request-ports> |
    "data-feed/" <data-feed-ports>

<service-request-ports> ::= "request/" <entity-name> "/" <service-name> | "input" |
    "output/" <entity-name> "/" <client-service-name> | "reply/" <entity-name> "/"
    <service-name>

<data-feed-ports> ::= "output/" <data-feed-name> | "input/" <entity-name> "/"
    <data-feed-component-name> "/" <data-feed-name>
```

The only new field introduced here is the `entity-name` field. This should be filled with the name of the robot (or machine) in which the Service is running.

4.1.8 Syntactic Sugar Framework

This framework is simply a placeholder for auxiliary classes and definitions that are used throughout the code that are useful in order to beautify the code (hence the name).

4.1.9 System Framework

In this framework we have classes that represent an abstraction of specific data types of the underlying computing platform. Its purpose is for Service developers to be able to use functionality that is normally available in any system while using platform independent code. Currently the only abstract class present in this framework is `Time`. This class allows developers to represent time in a platform-independent format. In a particular system this task is instantiated as a class compatible with the underlying system's time representation format.

4.1.10 XML Framework

All of MeRMaID::support's configuration files are written in XML and have corresponding XML DTD files for validation of their content. In order for these to be used in a platform independent way by Service developers, the XML Framework was created. This framework allows for XML files to be read and validated. This framework implements an approach similar to the Document Object Model (DOM) Level 1 Specification [2], although simplified. Since this is not a DOM Level 1-compliant implementation, not all of the possibilities of XML representation are available, but those that are available have met all the needs of the project until now.

If more functionality for XML manipulation is needed, this framework should be extended in order to support it.

4.1.11 Current Implementation

MeRMaID::support currently only uses three libraries: ACE for dealing with `ActiveObject` execution, libxml2 for reading and validating XML files and YARP for communication (which also uses ACE). The rest of the code is written using only standard C++ and C++'s Standard Template Library (STL). Both ACE and libxml2 are libraries that are available in a wide variety of systems and were chosen precisely because of that. Given this, MeRMaID::support is expected to run in all the platforms supported by both of these libraries. MeRMaID::support has been tested in both Linux and Mac OS X operating systems. Currently MeRMaID::support is not working on Windows platforms because of several differences between the GCC compiler used in Linux and Mac OS X and Microsoft's Visual Studio compiler which cause the code to not compile in Windows.

The build process of MeRMaID::support is managed by the CMake system which is able to generate build files for several platforms. CMake is configured by a series of script files which indicate how the code should be compiled and then generates system native build files. For instance, in a Unix environment, CMake generates regular makefiles that build the code, while in Windows it can generate Visual C++ projects. CMake is also available for a wide variety of systems.

4.1.12 Example on coding simple Services using MeRMaID::support

In this section an example of two simple Services that interact with each other is presented. The EchoService provides a Service Interface to which requests with any type of data may be sent via a Service Request. The EchoService then just simply sends the data received in the Service Request to the Service Reply, effectively echoing all data received.

Let us start by writing the Service Description file. The Service Description file states the Service's name, a human-readable comment on the Service's functionality and the description of all of its Service Interfaces. In this case we will have only one Service Interface named "EchoInterface". Since in this example we don't want MeRMaID::support to validate the structure of the data received in the Service Interface, these fields should be left empty in the Service Description file. The resulting XML can be seen in Listing 4.4.

Next we will build the header file for our Service. To build Services for MeRMaID::support the developer should inherit from the base Service class. The constructor for the class must follow the same signature as the base Service class: two arguments, being the first a pointer to the ActiveObject in which the Service is going to be run and the second a pointer to the XmlDocument containing the Service Description. Additionally to the constructor we will need a handler function where we will process the Service Requests that will arrive to the Service via the previously declared Service Interface in the XML Service Description file. The handler functions receives as arguments pointers to both the Service Request received and to the Service reply that will be sent back. The handler function should set the data in the Service Reply in accordance to what is expected to be replied when sending the given request to the Service Interface to which it is associated. The

Listing 4.4: echo-service-description.xml: EchoService description file

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE service-description SYSTEM "service-description.dtd">
<service-description>
  <name>EchoService</name>
  <comment>This services echoes to the reply the data that arrives in the
      input</comment>
  <service-interface>
    <name>EchoInterface</name>
    <comment>Echoes the request data back to the requester</comment>
    <in-data></in-data>
    <out-data></out-data>
  </service-interface>
</service-description>
```

header file with all of these declarations can be seen in Listing 4.5.

Listing 4.5: EchoService.hpp: EchoService declaration

```
namespace examples
{
  class EchoService : public Service
  {
    public:
      EchoService(CountedPtr<ActiveObject> ao, CountedPtr<XmlDocument>
          serviceDescription);

      static void echoHandler(CountedPtr<ServiceAsynchRequest> request,
          CountedPtr<ServiceAsynchReply> reply);
  }; // EchoService

} // namespace examples
```

Now, we will write the code that actually implements EchoService. We don't need to do any initializations in EchoService's constructor, so it will have an empty body, but, just like any other Service, it has to call the constructor of the base Service class. Other than the constructor the EchoService class only has a handler function that is responsible for processing the Service Request. In the EchoService this function only has to set the Service Reply data the same as the data that arrives in the Service Request. The complete code of the EchoService.cpp file containing both the constructor and the handler function is shown in Listing 4.6.

Having built the EchoService, now follows the construction of a simple Service that is able to interact with

Listing 4.6: EchoService.cpp: EchoService implementation

```
#include "EchoService.hpp"

using namespace examples;

EchoService::EchoService(CountedPtr<ActiveObject> ao, CountedPtr<XmlDocument>
    serviceDescription) : Service(ao, serviceDescription)
{
  registerServiceAsynchRequestHandler("EchoService", &echoHandler);
}; // EchoService()


void EchoService::echoHandler(CountedPtr<ServiceAsynchRequest> request,
    CountedPtr<ServiceAsynchReply> reply)
{
  //put service request data on a local variable
  CowPtr<DataValueVector> data = request->getRequestData();

  // set the reply data equal to the request data
  reply->setReplyData(data);

}; // echoHandler
```

it. EchoClientService simply runs in regular time intervals (say, one time per second) at which it sends data to the EchoService. It has no Service Interfaces, therefore its description file consists only of its name and a special field named "update-frequency" that specifies the frequency at which the Service's `update()` method should be invoked by MeRMaID::support. The Service Description XML for EchoClientService can be seen in listing 4.7.

Listing 4.7: echo-client-service-description.xml: EchoClientService description file

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE service-description SYSTEM "service-description.dtd">
<service-description>
  <name>EchoClientService</name>
  <comment>This services runs periodically and sends data to the
      EchoService</comment>
  <update-frequency>1</update-frequency>
</service-description>
```

Following the same sequence presented for the EchoService, now we'll write the EchoClientService.hpp header file. The header file for EchoClientService is in all similar to the header for EchoService, except that the handler is for receiving a Service Reply instead of a Service Request and the EchoClientService class implements the `update()` method. The complete header code can be found in Listing 4.8.

Listing 4.8: EchoClientService.hpp: EchoClientService declaration

```
namespace examples
{
  class EchoClientService : public Service
  {
    public:

      EchoClientService(CountedPtr<ActiveObject> ao, CountedPtr<XmlDocument>
          serviceDescription);

      static void EchoServiceReplyHandler(CountedPtr<ServiceAsynchReply> reply);

      virtual void update();
  }; // EchoClientService

} // namespace examples
```

Finally, the code that implements EchoClientService. Just like EchoService, we don't need to do anything in the constructor except for calling the base-class (Service) constructor. The `update()` method is where we will send request to EchoService. Having defined the "update-frequency" in EchoClientService's Service

Description file, this method is called automatically at the frequency written in that file. To make a new ServiceRequest, we ask the Service class to generate a new one for us via the `makeNewServiceAsynchRequest()` method. This method has one argument which is the name of the Service to which the request will be sent. After having a new Service Request generated, we put some data in it. In this case we simply insert a small string: "ECHO!". All the data manipulation mechanisms are provided by the Data Framework described in section 4.1.3. We put this data inside the Service Request and we send it via the `sendServiceAsynchRequest` method, passing as parameters the name of the destination Service Interface, the Service Request and a pointer to a handler function for the Service Reply. Finally, we have to code the handler for processing the Service Reply that will later be received in response to the Service Request sent. In this handler we simply print the Service Request ID number of the originating Service Request (every Service Request sent by a Service is given a unique ID number, so that it may know which was the originating request of the reply). The code can be seen in Listing 4.9.

4.2 MeRMaID

The reason for building the middleware layer presented in the previous section called MeRMaID::support was to be possible to implement platform-independent high-level components for MeRMaID (in the form of Services). Having a clear definition (and implementation) of Services is still not enough to guarantee that it is possible to have a modular architecture in which it is easy to replace a certain Service by another one that is able to perform a certain task. Therefore, MeRMaID defines a high-level architecture design (as described in [14]), based on Services and how they should interact with each other. As an analogy, MeRMaID::support can be seen as providing the building blocks (Services) and MeRMaID provides the plan of how to assemble them in order to build a sound system. Using this "plan" that MeRMaID provides, it is possible to substitute a block (Service) by another one, as long as the requirements of the "plan" are followed.

An important aspect that should be noted is that while MeRMaID::support is completely domain-independent, not having any direct link to robotic applications (although requirements inherent to these applications were, of course, accounted for in this level), MeRMaID is completely dedicated in describing how Services should be organized in the domain of cooperative robotics. In fact, MeRMaID can be seen as a Domain-Specific Software Architecture (DSSA), i.e. "an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, composed in a standardized structure (topology) effective for building successful applications" [19].

MeRMaID is not currently implemented with the previously described version of MeRMaID::support, but with an older version that didn't have such a clear structure. Nevertheless, the structure presented in this section is completely valid and independent of the implementation of MeRMaID::support.

4.2.1 MeRMaID High-Level Concepts

The design of MeRMaID started by the definition of key-concepts that were result of several years of experience with soccer robots within the SocRob project [6][12] (http://socrob.isr.ist.utl.pt) at ISR. These concepts are a way of partitioning the problem of controlling a (team of) robot(s). This is why MeRMaID is so tightly

Listing 4.9: EchoClientService.cpp: EchoClientService implementation

```
#include "EchoClientService.hpp"

using namespace examples;

EchoClientService::EchoClientService(CountedPtr<ActiveObject> ao,
    CountedPtr<XmlDocument> serviceDescription) : Service(ao, serviceDescription)
{
  // nothing to be done apart from calling Service's constructor
}; // EchoCEchoClientService()

void EchoClientService::update()
{
    CountedPtr<ServiceAsynchRequest> sr =
        makeNewServiceAsynchRequest("EchoService"); // get a new service request

  CowPtr<DataValueVector> data = sr->getRequestData(); // get data vector for this
      request
  CowPtr<DataValue> s = DataFactory::buildString("ECHO!"); // build a data string
  v->pushBack(s); // put the string in the data vector
  sr->setRequestData(data); // put the data vector back in the request

  sendServiceAsynchRequest("EchoInterface", sr, &EchoServiceReplyHandler); // send
      the request

}; // run()

void EchoClientService::EchoServiceReplyHandler(CountedPtr<ServiceAsynchReply>
    reply)
{
  CowPtr<DataValueVector> data = sr->getReplyData();
  std::cout << "Received:␣" <<  reply for request #"␣<<␣reqId␣<<␣std::endl;
};␣//␣EchoServiceReplyHandler()
```

attached to the robotics domain: because it builds on concepts that are mostly applicable only to robotic applications. These concepts are:

Navigation Primitive A Navigation Primitive is a guidance algorithm which, based on the current and target robot postures and current self-localization estimate, computes the required actuator commands to move the robot from the current to the target position avoiding obstacles on the way.

Primitive Action A Primitive Action is the atomic element of a behavior, which can not be further decomposed. It usually consists of come calculations (e.g., determination of the desired posture) plus a call to a navigation primitive or the direct activation of an actuator. Desirably it is designed as a STA (Sense-Think-Act) loop, i.e., a generalized view of the closed-loop control system concept. This means that MeRMaID favors a Primitive Action that moves the robot towards its goal while avoiding obstacles, rather than having one primitive that moves towards the goal and another that avoids obstacles.

Behaviors Behaviors are defined as "macros" of Primitive Actions grouped together using some appropriate representation. For instance, a behavior may consist of a state machine in which states represent Primitive Actions and transitions between states have associated Events, but it could also be defined by a fuzzy decision-making algorithm based on fuzzy rules, used to select sequences of Primitive Actions to be executed.

Predicates Predicates are boolean relations over the domain of world objects, e.g., see(x), where x can be ball, pole, or field_line, in the soccer domain, or near(r,x), where r is any of the team robots, and x can be any world object.

Events An Event is, in general, an instantaneous occurrence which denotes a state change (e.g., of a variable, of a robot). In MeRMaID, we limit the event definition to changes of (logical conditions over) Predicates from `True` to `False` or `False` to `True`. Examples of events in robot soccer are: event `lost_object` occurs when the predicate `has`(object) changes its value from `True` to `False`, and vice-versa for event `got_object`; event `found_object` occurs when the predicate `see`(object) changes from `False` to `True`.

Roles Roles are subsets of behaviors, defined over the set of available behaviors. When a role is selected (e.g., `Attacker`, `Defender`, `GoalKeeper` in the soccer domain), a new set of behaviors becomes enabled for selection by the behavior coordination mechanism. In practice, a role constrains the possible options for a robot selection of behaviors, effectively constraining the overall behavior displayed by the robot. Note that roles do not form a partition over the set of available behaviors, since there are behaviors that may be shared by more than one role (e.g., `GetClose2Ball` for the `Attacker` and `Defender` roles above).

4.2.2 MeRMaID's Structural Blocks

MeRMaID is divided in three major structural blocks: Atlas, Wisdom and CORTEX. A diagram with the structure of MeRMaID's high level architecture can be seen in Figure 4.5

The three structural blocks of MeRMaID are described in the following subsections.

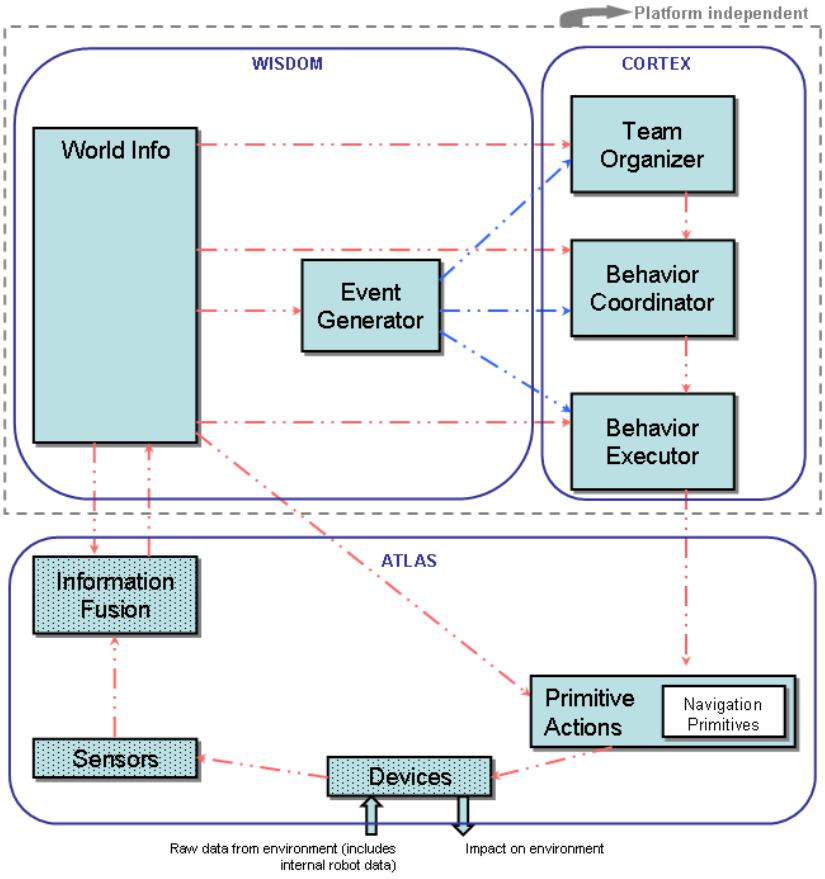

Figure 4.5: MeRMaID High Level Architecture: red lines are data connections, blue lines are event connections

4.2.2.1 Atlas

Atlas is viewed as the subsystem that supports the whole system. It is responsible for the tasks most directly related to the robot's environment: sensing and acting. The Atlas block is different from the other blocks in the sense that it does not specify individual Services, but types of Services, being there more than one instance of each Service. This block is composed by the following types of services:

Devices handle the low level interface with physical-world devices (e.g. motors, sonars, cameras).

Sensors obtain information from the devices (e.g. odometry, obstacle location, ball position).

Information Fusion fuse information from several sensors (which can be sensors onboard the robot or from external sources).

Primitive Actions see definition in the previous section.

Navigation Primitives see definition in the previous section.

4.2.2.2 Wisdom

This block is named Wisdom in order to denote that it is a very relevant requirement for intelligence to be displayed. This block acts as a central point of information storage and has the ability to generate events based on predicate changes. This block is home for the following Services:

World Info stores general purpose high-level data, relevant for predicate evaluation. World Info may hold information originating from other external sources of information like other robots or external sensors.

Event Generator generates events based on predicate changes. This service handles all the available predicates and re-evaluates them when every time relevant data changes in World Info (i.e. while a certain predicate's input data is not changed, it will not be re-evaluated in order to avoid unnecessary processing). If a predicate's value changes the Event Generator generates an event that is automatically sent to all Services that registered for it previously.

4.2.2.3 CORTEX

The name CORTEX comes from CoORdinator, TEam organizer and eXecutor. This block is responsible for decision-making and is the one which effectively controls the behavior of the robotic system. This block is composed by three Services from which its name originated:

Team Organizer is responsible for the actual organization of the team in terms of roles. It activates roles in each of the team's robots.

Behavior Coordinator is responsible for behavior selection and coordination. It activates a behavior from the set of behaviors available for the currently selected role.

Behavior Executor is responsible for behavior execution. It activates Primitive Actions, for the currently selected behavior.

These three services should be implemented using some kind of formal behavior selection mechanism. Currently there are implementation for finite state machines, Petri net plans and fuzzy logic decision-making-based behavior selection.

4.3 mlgen: MeRMaID Loader Generator

MeRMaID::support is intended to provide platform-independence and all the Services present in MeRMaID written in a platform independent way (except those that interact with hardware, of course). However, there is still one aspect remaining for full platform independence: the deployment of the Services. To address this issue the MeRMaID Loader generator, or mlgen for short, was created. This utility program is able to generate code that deploys the Services based on an XML description of the Services to deploy.

The XML file for describing the runtime configuration of the Services to be deployed declares a number of ActiveObjects and Services that should be run. For each ActiveObject the only information necessary is a name for identifying it. For each Service the following information is needed:

service description file The filename that contains the Service Description.

header file The .hpp file of the specific Service implementation.

namespace The C++ namespace in which the Service was declared.

class name The C++ class name given to the Service's concrete implementation.

instance name A name for identifying the instance of the Service.

configuration file A configuration filename to be read by the Service when it is instantiated.

ActiveObject name The name of the ActiveObject in which this Service should be run.

Currently mlgen is able to generate code for deploying MeRMaID Services for the ACE-based implementation of MeRMaID::support. In listing 4.10, a sample file for configuring mlgen can be seen. This configuration file for mlgen is based on the example given in section 4.1.12. In this example we have two ActiveObjects named "ActiveObject1" and "ActiveObject2" and two Services: "EchoService" and "EchoClientService". Each Service is set to run in one ActiveObject so that the Services may run concurrently. If the Services could not be run concurrently because of, for instance, a shared resource, they should be run in the same ActiveObject. Since there is nothing limiting the Services to run concurrently, we may put them on different ActiveObjects and gain some performance from it. By running mlgen with this configuration and then compiling the resulting output code together with the code of EchoService and EchoClientService we would get an executable program that deploys and starts execution of both of the Services.

4.4 SIF: SocRob Interface

The SocRob Interface, or SIF for short, was developed in response to certain specific needs of the SocRob project. In robotic soccer, the teams need to have a so-called base-station. This base-station is typically a PC connected by an ethernet connection to the computer that is running the referee-box (a special software that allows the referee to control the state of the game). In the base-station, teams should run a software that connects to the referee box and that relays the game state information to the team's robots. the SocRob Interface acts as this relay during soccer games while giving information about the robots on the screen.

The Java programming language was chosen to develop SIF. The main reason for choosing Java was that it is supported in a wide number of different Operating Systems and offers a reasonable framework for designing interfaces.

4.4.1 User Interface

The user interface is divided in several tabs, each tab having its own layout. By default, SIF starts with a tab showing the Team Layout. In this layout, a general view of the status of the robotic team can be seen. This layout is dominated by the field view in which the user may see the robots' postures and other sensing information like their perceived position of the goals and the ball. There are also controls for connecting to the referee box and for connecting to the robots. A screenshot of this layout can be seen in Figure 4.6. The other layouts available in other tabs are all similar and display robot-specific information, i.e., each tab displays information about only one robot. In these layouts there is information about the robots sonars, batteries and a text field to which various information about the communication with robot is printed (like connections and disconnections, sent events, etc.). A screenshot of this layout is seen in Figure 4.7.

4.4.2 Internal Code Structure

SIF's code follows the model-view-controller architectural pattern [10], i.e., the domain objects and their graphical representation and interaction mechanisms are separated in order to minimize the dependencies between these aspects. This way, SIF has a collection of domain objects that together represent the state of all objects

Listing 4.10: Example configuration file for mlgen.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE runtime-configuration SYSTEM "runtime-configuration.dtd">
<runtime-configuration>
  <active-object>
    <active-object-name>ActiveObject1</active-object-name>
  </active-object>
  <active-object>
    <active-object-name>ActiveObject2</active-object-name>
  </active-object>
  <service>
    <service-description-file>echo-service-description.xml</service-description-file>
    <header-file>EchoService.hpp</header-file>
    <namespace>examples</namespace>
    <class-name>EchoService</class-name>
    <instance-name>echoServiceInstance</instance-name>
    <configuration-file></configuration-file>
    <active-object-name>ActiveObject1</active-object-name>
  </service>
  <service>
    <service-description-file>echo-client-service-description.xml
        </service-description-file>
    <header-file>EchoClientService.hpp</header-file>
    <namespace>examples</namespace>
    <class-name>EchoClientService</class-name>
    <instance-name>echoClientServiceInstance</instance-name>
    <configuration-file></configuration-file>
    <active-object-name>ActiveObject2</active-object-name>
  </service>
</runtime-configuration>
```

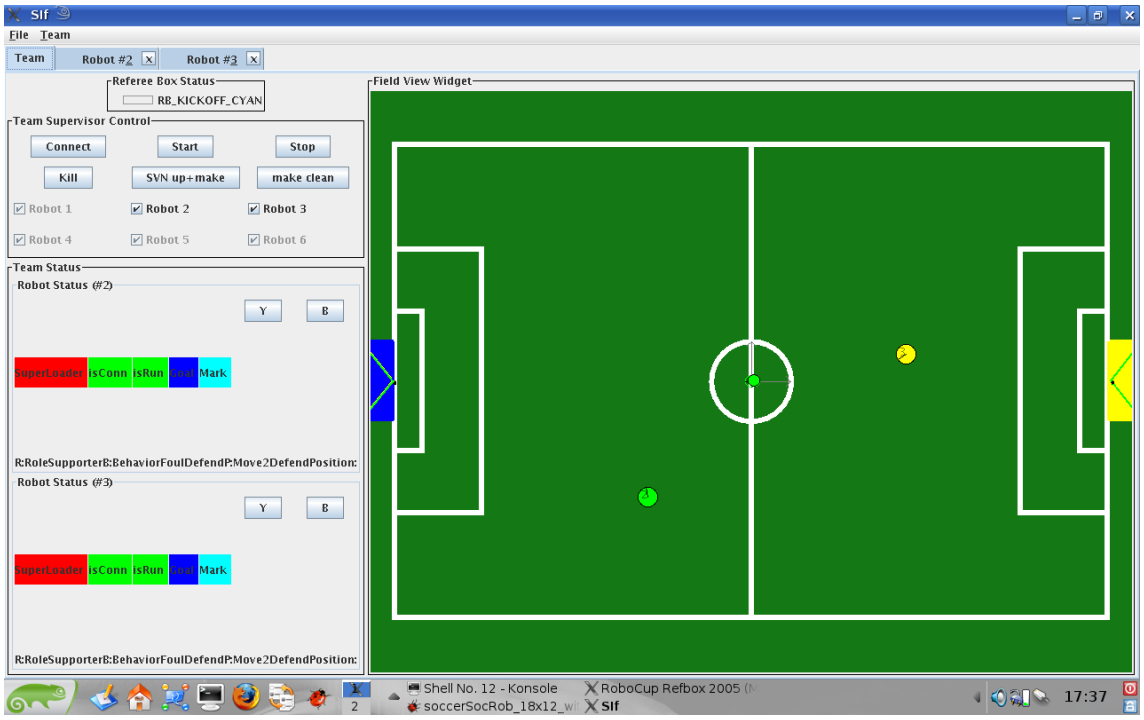

Figure 4.6: SIF's Team Layout

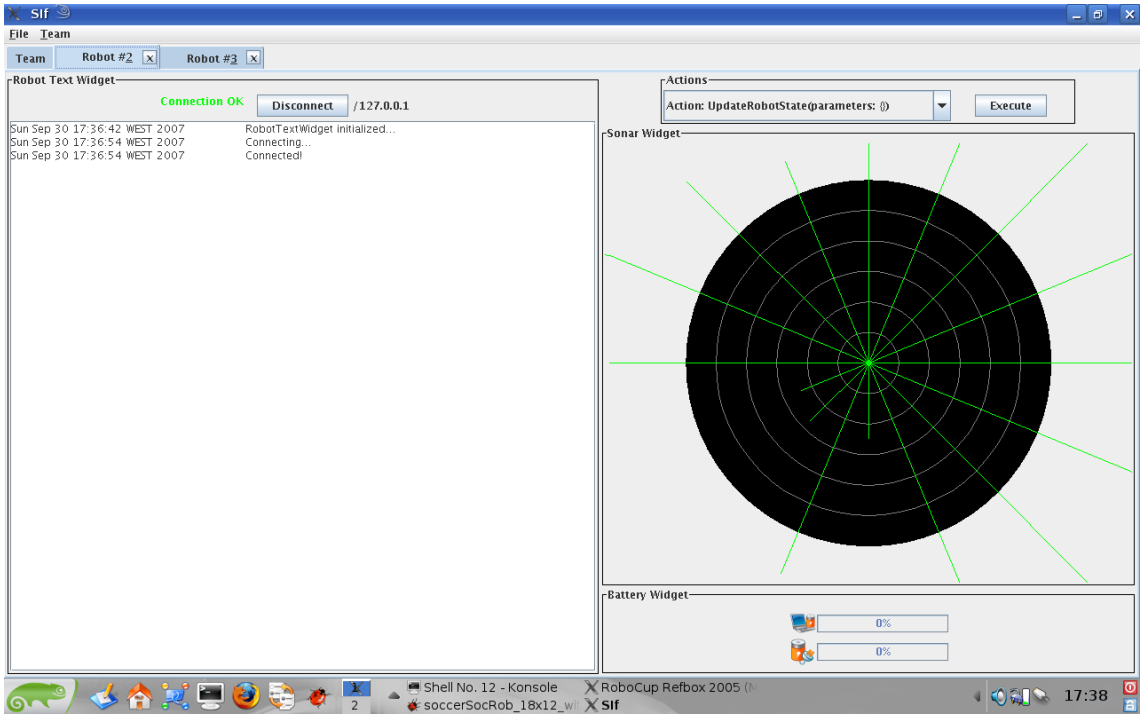

Figure 4.7: SIF's Robot Layout

on which it has information. The information for updating these domain objects arrives through network connections to the robots and to the referee box. The domain objects are all in the same Java package named `sif.realworld`. The objects responsible for showing information to the user (the `components`) get the required information from the domain objects so they are decoupled from how (and when) the information is received from the robots or the referee-box. These `components` are then assembled in small groups called `widgets`. These are simple collections of `components` that provide information to the user in some kind of coherent format. For instance, the `FieldViewWidget` is composed by a `Field component` which draws the field representation, on top of which it puts `RobotAvatar components` representing the robots.

The code is organized in several Java packages and they contain the following:

sif Thread management and event diffusion mechanism among objects.

sif.communication Implementation of the communication protocols with the robots and the RoboCup MSL Referee Box.

sif.datatypes Datatypes used throughout the code (such as `Posture` and `Obstacle`)

sif.layouts Implementation of the GUI layouts. Currently the `TeamLayout` and the `RobotLayout` are implemented.

sif.menu Holds the application menu classes.

sif.menu.items Implementation of the menu items.

sif.realworld Domain objects: `Ball`, `Goal`, `Robot` and `Team`

sif.widgets Implementation of the widgets used in the layouts.

sif.widgets.components Implementation of the components used inside widgets.

4.4.3 Integration with MeRMaID

The SocRob project already had a GUI for displaying information from the robots and for connecting to the referee-box, named GameIface This interface had been developed without a structured software development approach and poor documentation, so code maintenance and development became increasingly more difficult. Nevertheless, GameIface was able to perform its task and being a fundamental part for putting the robotic team working with the referee-box it could not be abandoned instantly. Furthermore, since it was expected for the development of SIF to take some time until the code got stable enough to be used in real games, we had to make a solution that would enable developers to transition gradually from one interface to another and have always GameIface as a fail-safe backup option. Therefore, it was decided to develop SIF communication to the robots using the same protocol that was being used by GameIface.

The existing protocol consisted on a simple request-response message exchange pattern working over TCP/IP. There are two types of requests: a getter request which gets information values from the robot and a setter request which sets information values in the robot.

The messages are sent in ASCII text, being each message terminated by a newline ASCII character. The getter request follows the format "`GETV` <*value name*>", in which <*value name*> is the name of the value being requested. The response to a getter request is structured like "`+OK` <*value*>", with <*value*>, being a text-representation of the requested value. If an error occurs during the processing of the getter request the message "`-ERR`" is returned instead. As for the setter request, it follows a similar format with the addition that

the value is also sent in the request: "SETV <*value name*> # <*value*>". On success, the response is simply "+OK" or if there was an error "-ERR" is sent.

The value names of the available data are related to the old names used in the previous software architecture used in the SocRob project. A list of the used names and the meaning of its data is listed in Table 4.2.

This text-based request-response protocol is implemented in SIF in the `sif.communication` package and is used by the domain-objects to get data from the robots. In MeRMaID, since this protocol does not conform to the protocol used between MeRMaID services, a Service named `CommunicationManager` was developed to handle communication between the robots and SIF (as well as the old GameIface which followed the same protocol). This Service is able to handle the GUIs' requests and translate them into requests understandable by other relevant MeRMaID Services.

Value Name	Description
local.odometry.x	Robot's posture x component.
local.odometry.y	Robot's posture y component.
local.odometry.theta	Robot's posture theta component.
local.sonar*N*.dist	Distance measured by sonar number *N*.
local.vision.ball.x	Locally viewed ball's posture x component.
local.vision.ball.y	Locally viewed ball's posture y component.
global.ball.x	Global (data-fused) ball's posture x component.
global.ball.y	Global (data-fused) ball's posture x component.
local.vision.ball.see-ball?	Boolean indicating if ball is locally visible or not.
global.ball.see-ball?	Boolean indicating if ball is globally visible or not.
BatteryLaptopState	State of the robot's laptop battery.
BatteryRobotState	State of the robot's batteries.
local.vision.yellowgoal.x	Yellow goal's posture x component.
local.vision.yellowgoal.y	Yellow goal's posture y component.
local.vision.yellowgoal.theta	Yellow goal's posture theta component.
local.vision.yellowgoal.see-goal?	Boolean indicating if yellow goal is visible.
local.vision.bluegoal.x	Blue goal's posture x component.
local.vision.bluegoal.y	Blue goal's posture y component.
local.vision.bluegoal.theta	Blue goal's posture theta component.
local.vision.bluegoal.see-goal?	Boolean indicating if blue goal is visible.
CurrentRole	The current role being performed by the robot.
RunningBehavior	The current behavior being executed by the robot.
RunningPrimitiveActions	The set of currently running Primitive Actions.
isRunning	Boolean indicating if the robot is running (i.e. all electronics on and ready to move).
refevent	Value name for setting the last referee event in the robot.
setposture	Value name for setting a global posture in the robot.

Table 4.2: Value names used in the text-based communication protocol between GUIs and robots

5 Integration with other Tools

MeRMaID has already been integrated with other tools in order to support the development of robotic systems and the design of their behavior. In this section three tools are described as well as how they are integrated with MeRMaID.

5.1 FSMeditor

FSMeditor is a visual editor for Finite State Machines (FSMs). Users are able to visually edit a representation of a FSM and then load it to a FSM implementation of MeRMaID's CORTEX components.

5.2 JARP

JARP's purpose is identical to that of FSMeditor but for Petri Nets. Users visually edit a Petri Net, save it to a Petri Net Markup Language (PNML) file that can be read by an implementation of MeRMaID CORTEX components that are able to read these files. This way, developers may develop full behaviors at all three levels of CORTEX using a visual tool. In figure 5.1 an example of a Petri Net being edited in JARP can be seen.

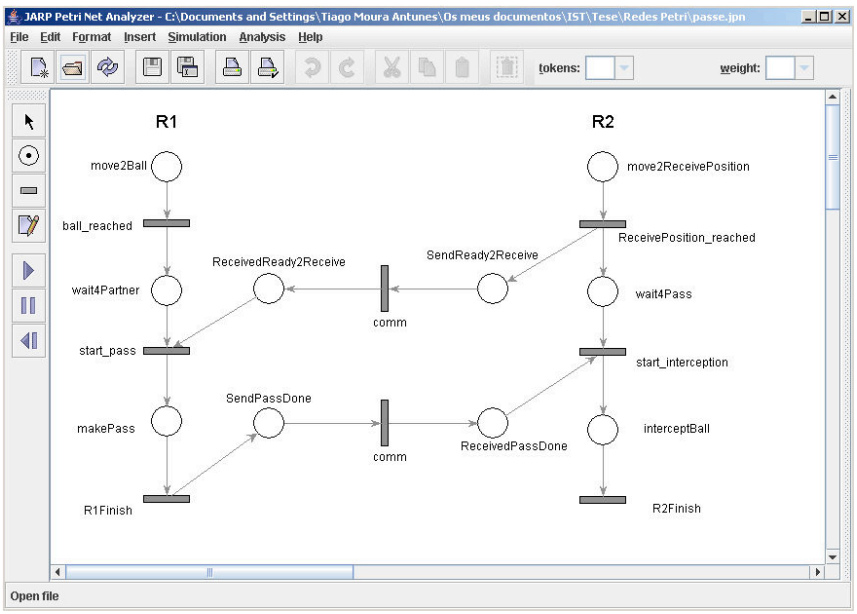

Figure 5.1: JARP being used to develop a petri-net

5.3 Webots

Webots is a commercial robot simulator made by Cyberbotics (http://www.cyberbotics.com). It has been used in the SocRob project in order to test and debug the robots' control code in a controlled environment, where experiments can be quickly repeated and are free from robot hardware failure. The integration of Webots to MeRMaID was fairly easy: all it took was to substitute MeRMaID's Device Services from those that interface

with the robots' physical sensors and actuators with Device Services that would interface with Webots. In figure 5.2 two full teams of robots ready for playing a robotic soccer match can be seen.

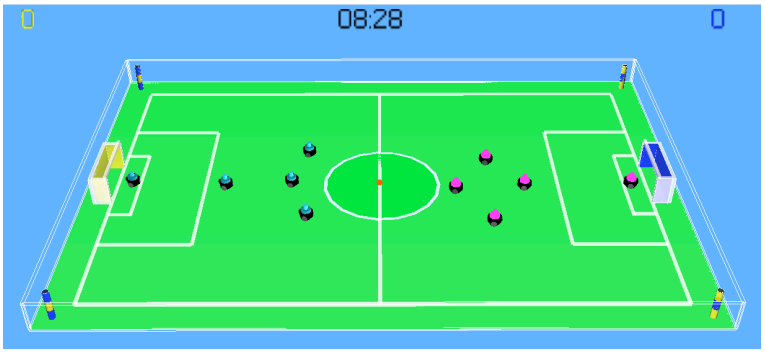

Figure 5.2: Webots simulator used for testing.

6 Solution Application and Results

6.1 SIF: SocRob Interface in use during games and development

SIF was extensively used during the preparation for RoboCup'07 as well as during the competition games. RoboCup and, more specifically, its Middle-Size League is a robotic soccer competition in which teams of fully autonomous robots compete with each other in an environment and with a set of rules in all similar to human soccer. The main objective of this type of competition is to foster investigation in several important areas for the development of robotics in a scenario that is attractive to the general public.

Besides the field robots, there is one external machine for each team that typically runs the teams interface and, most importantly, software to relay information of the game state arriving from a PC where the team of referees issues its rulings. This is the only way the robots can know if the game is running or not, or if a foul was issued by the referees. SIF had the responsibility of doing the relay of this information to the robots. It was a critical component of all the system since failure in working would result in all the robotic team failing. From the visualization side, although SIF doesn't provide many new features compared to the previous GUI used in the SocRob project, the additional information it displays was useful in debugging problems and to understand how the whole system was working (namely information about the localization of the field goals from the robot's perspective). On the other hand, it was expected for SIF's performance to be better than what it displayed, being the application not as responsive as desired. This was caused mainly because of the communication protocol used and the very high latencies in communication at the RoboCup venue, where packet round-trips to and from the robots can easily be in the order of several hundreds of milliseconds, and sometimes can become as high as several seconds.

6.2 MeRMaID::support: application in the URUS project

A demo application was prepared in the context of the URUS project. The general objective of the URUS project is to develop "new ways of cooperation between network robots and human beings and/or the environment in urban areas" [17]. The project has 11 partners, each participating in several work-packages dedicated to different aspects of the overall system. Therefore, an important issue arises: integrating all the work developed by the different partners. The communication protocol described in Section 4.1.7 was accepted as the communication protocol to be used between the software packages. Thus, MeRMaID::support is already compatible with the protocol and ready to be used in this project to integrate several software packages.

The main goal of the demo here presented was to demonstrate a fully working deployment of all the software based on MeRMaID::support and using the defined communication protocol based on YARP. The demo involved controlling a complete robotic application with both fixed sensors and a mobile robot with on-board sensors as well.

The task to be accomplished by the system was to identify special markers placed in the environment and to localize them. Due to the low precision that is given by the fixed camera, the mobile robot should move nearer to the markers and improve their localization estimate. As the robot did not have any sensor capable of detecting the object, the demo was simplified to make the robot just go near to the target object.

6.2.1 Demo Layout

The demo consisted of:

- one USB camera connected to a PC
- one mobile robot (Pioneer) with an on-board PC
- wireless network infrastructure based on the 802.11 family of protocols

A diagram of the layout used can be seen in Figure 6.1.

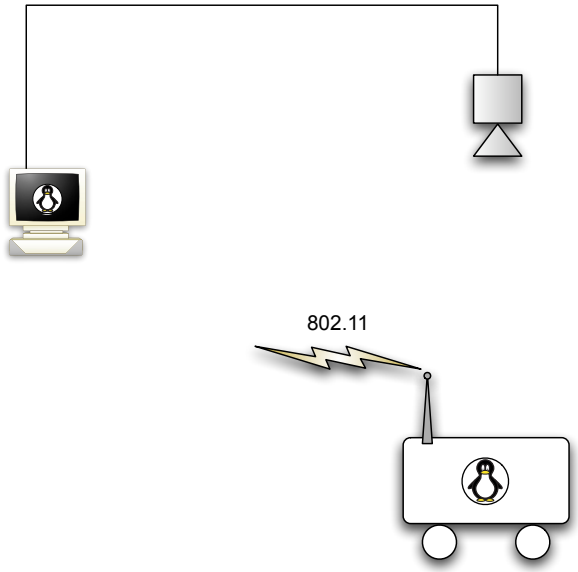

Figure 6.1: URUS demo layout diagram

6.2.2 Porting existing code to MeRMaID::support

The code for this system was already fully developed by the time MeRMaID::support was ready. It was using YARP directly and therefore needed to be ported to MeRMaID::support and had been developed without any knowledge that it would be later be used with MeRMaID::support. The porting process started by identifying the communication links between software components and asserting that they could be classified as Service Request or Data Feed communications. The identification of the communication links and software components led to a straightforward definition of how the software components capabilities would be mapped to MeRMaID::support Service's as well as the Asynchronous Service Request and Data Feed communication methods to be used.

6.2.3 Software Structure

A Service built on MeRMaID::support was built for each key functionality of the System. In Figure 6.2 the software structure of the demo can be seen. The Services created for this demo were:

`PioneerControl` This Service controls directly the Pioneer robot in which it is running. It is able to set the robot's wheel velocities and to export data from the sonar sensors and from the odometry encoders.

FeatureDetector This Service is able to connect to a USB camera and extract the position in world coordinates of the robot and of the target object.

RobotPostureEstimator This Service collects data from the `PioneerControl` Service (the robot's odometry encoders) and from the `FeatureDetector` Service (the location of the robot) and fuses it in order to provide accurate estimates of the robot's posture.

RobotController This Service gathers information from the `RobotPostureEstimator` (the robot's posture) and is able to guide the robot to a specific posture. The posture to which it drives the robot is set through the Service Interfaces it provides: "gotoPosture" and "stop".

GlobalController This Service has the main control logic for implementing the demo. It gets information from the detected postures of the robot and the target object from the `FeatureDetector` Service. If the robot is seen too far away of the target object, it orders the robot to go to a posture near the object by sending a Service Request to the `RobotController`'s `gotoPosture` Service Interface.

Service Requests are used primarily for controlling the behavior of Services. As can be seen, the `GlobalController` controls the `RobotController` that in turn controls the robot (`PioneerControl` Service). Data Feeds are mainly used as means for sharing data between Services.

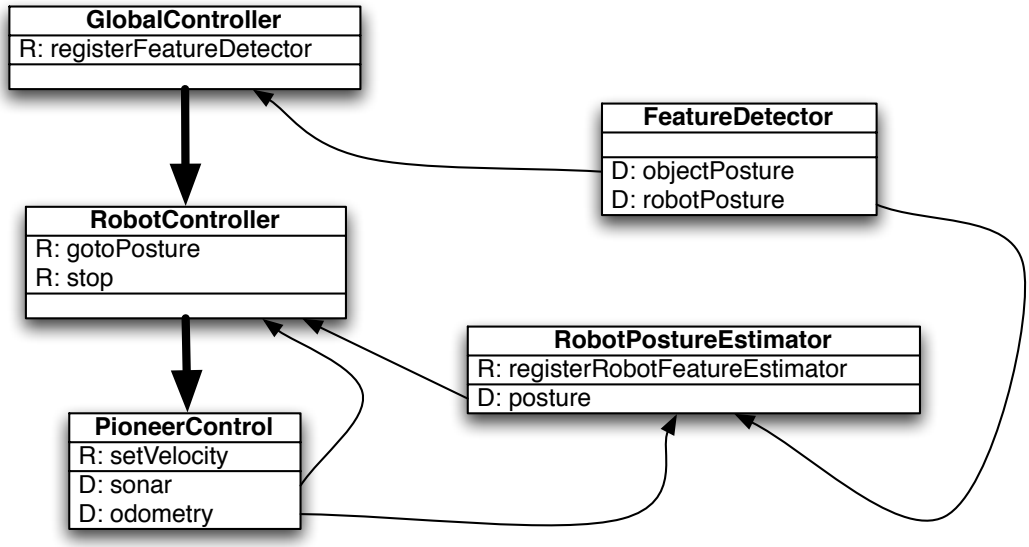

Figure 6.2: URUS demo software structure. Each box is a Service and has the available Service Interfaces indicated by an 'R' and the Data Feeds provided by each Service indicated with a 'D'. The big arrows indicate control flow, and the thin arrows indicate data flow.

6.2.4 Results

Integration of all the software components was successful and conformance to using MeRMaID::support-based Services was verified. This demo did not base itself on the MeRMaID high-level architecture but was an example of how MeRMaID::support and its Service-based approach to separate software components can be applied.

7 Conclusions

7.1 Main Contributions

A new version of MeRMaID::support was designed and implemented using the Service-Oriented Architecture (SOA) Reference Model as a guide for structuring the solution. MeRMaID::support also relies on the Active Object design pattern and implements it using the ACE library which is widely used in the industry. MeRMaID::support was built in order to ease the developer's task of developing services and to interact with other services. The various frameworks available in MeRMaID::support are targeted towards specific needs and specific requirements of service developers.

A communication protocol built on top of the commonly used YARP library was developed. The simplicity and clear definition of this protocol enables software components built without using MeRMaID::support to be able to interact with MeRMaID::support-based Services.

A tool for deploying MeRMaID::support services, mlgen, was developed, further increasing the platform-independence of the environment in which the Service developer works. By also specifying MeRMaID::support's building process using CMake scripts, platform independence is achieved for the whole process from building MeRMaID::support-based Services to deploying them.

A special GUI for aiding development of SocRob's project software, the SocRob Interface (SIF), was developed. This GUI provided visual feedback of the inner workings of MeRMaID Services as well as serving an important function during robotic soccer competition matches in which it acted also as a base station for game state reporting to the robots.

7.2 Future Work

As for future work that should follow what has been described, the following is suggested:

- Port SocRob's code to the newly developed version of MeRMaID::support: porting the code to the new version would allow all Services to communicate using a more structured communication model and allow for easier communication between robots, something that was coded manually using the previous version of MeRMaID::support.
- Implement more advanced mechanisms for invoking services such as service searching capabilities as well as automatic servicing by type of service (i.e. search for services based on their semantic function).
- Implement semantics-based data diffusion mechanism that would enable automated diffusion of data about a certain concept, even though it may come from different origins.
- Change SIF's communication protocol to be compatible with the communication protocol used between Services. It would also be useful that this communication could be implemented using UDP as the transport protocol in order to minimize latency (even though delivery is not guaranteed but that is not an important requisite for SIF).

Bibliography

[1] Microsoft Robotics Studio. http://msdn.microsoft.com/robotics/.

[2] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A.L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, et al. Document Object Model (DOM) Level 1 Specification. *W3C Recommendation http://www. w3. org/TR/1998/REC-DOM-Level-1-19981001, World Wide Web Consortium*, 1998.

[3] Marco Barbosa, Nelson Ramos, and Pedro Lima. MeRMaID - Multiple-Robot Middleware for Intelligent Decision-Making. *Proceedings of IAV2007 - 6th IFAC Symposium on Intelligent Autonomous Vehicles*, 2007.

[4] H. Bruyninckx. Open Robot Control Software: the OROCOS project. *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, 2001.

[5] D. Comer and D.L. Stevens. *Internetworking with TCP/IP*. Prentice Hall, 1999.

[6] Hugo Costelha, Nelson Ramos, Jo ao Estilita, Jo ao Santos, Matteo Tajana, Jo ao Torres, Tiago Antunes, and Pedro Lima. ISocRob-MSL 2007 Team Description Paper. Technical report, Instituto de Sistemas e Robótica - Instituto Superior Técnico, 2007.

[7] E. W. Dijkstra. *Co-operating Sequential Processes*. Academic Press, 1965.

[8] RT Fielding. *Representational state transfer (REST). Chapter 5 in Architectural Styles and the Design of Networkbased Software Architectures*. PhD thesis, University of California, Irvine, CA, 2000.

[9] B. Gerkey, R.T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[10] G.E. Krasner and S.T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[11] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. pages 483–499, 1996.

[12] Pedro Lima, Marco Barbosa, Jo ao Esteves, Nuno Lopes, Vasco d'Orey, and Hugo Pereira. ISocRob-4LL 2006 Team Description Paper. Technical report, Instituto de Sistemas e Robótica - Instituto Superior Técnico, 2006.

[13] G. Metta, P. Fitzpatrick, and L. Natale. YARP: yet another robot platform. *International Journal on Advanced Robotics Systems Special Issue on Software Development and Integration in Robotics*, 3:43–48, 2005.

[14] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *Software, IEEE*, 14:43–52, 1997.

[15] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on Standardization in Mobile Robot Programming :The Carnegie Mellon Navigation (CARMEN) Toolkit. *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, 2003.

[16] OASIS. Reference Model for Service Oriented Architecture 1.0. Technical report, OASIS, August 2006.

[17] A. Sanfeliu and J. Andrade. Ubiquitous Networking Robotics in Urban Settings. *IROS Workshop on Network Robot Systems*, pages 14–18, 2006.

[18] R. Simmons. Inter Process Communication (IPC). http://www.cs.cmu.edu/afs/cs/project/TCA/www/ipc/ipc.html.

[19] Will Tracz. DSSA (Domain-Specific Software Architecture): pedagogical example. *SIGSOFT Softw. Eng. Notes*, 20:49–62, 1995.

[20] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *Robotics and Automation, IEEE Transactions on*, 18(4):493–497, August 2002.

[21] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. *Aerospace Conference, 2001, IEEE Proceedings*, 1:121–132, 2001.