Università di Pisa

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Magistrale in Informatica

Tesi di laurea

iClips: an interaction system for controlling a robotic FACE

Candidato:
Nadia Vetrano

Relatore:
Antonio Cisternino

Correlatore:
Nicole Lazzeri

Controrelatore:
Giuseppe Prencipe

Anno Accademico 2011-2012

Alla mia famiglia e a Marco.

Contents

List of Figures

Listings

Abstract

Expert systems have reached large popularity since they were born in the 1980s. They are systems which encode human expertise in limited domains by representing human knowledge using if-then rules. Since one of the goals of Artificial Intelligence is to develop systems which exhibit *"intelligent"* human-like behaviour, expert systems fully find their way in this field.
iClips, the system developed in this work, is an interaction tool for controlling a robotic FACE. It takes its name from the existing Clips, an expert system developed from the NASA Space Centre that provides a complete environment for the construction of rule-based expert systems. iClips aims to extend the existing robotic project by implementing the cognitive part of the android through the introduction of a modular architecture which allows the user to coordinate, control and debug it.
Starting from a monolithic library, one of the goals iClips manages to reach is to build a deliberative/reactive two-levels architecture for coordinating the system and having a symbolic representation of our targets and purposes. By using symbolic abstraction iClips turns out to be a declarative tool. Due to this, it can be easily used by *hybrid subjects*, that means subjects who don't have very specific computer skills: while building a new module requires a programmer, working with iClips requires people who are just able to write simple logic rules.

Chapter 1

INTRODUCTION

The FACE project aims to realize a humanoid robot capable of expressing and conveying emotions and empathy. Its main goal is to enable autistic subjects to improve how they deal with emotional and expressive information, through familiarity and contextual information presented in a stepwise way. Before this work, the main problems with the project were that it was "computer science driven" and that the robot was controlled by a monolithic library. This was not a major issue during the early days of development, but when we decided to extend the library with new functionalities and behaviours, the monolithic structure came out to be a problem.
So we decided to redesign it and make it a declarative and modular system, which provides several advantages. The main tool we used to reach our goal was the Clips expert system. Expert systems are programs which encode human expertise in limited domains by representing knowledge using if-then rules.
A brief description of the structure of the thesis and of its chapters is as follows.

Chapter 2 (TOOLS and APPLICATION DOMAIN), as the title suggests, contains a description of the tools we've been using in this work and their application domain, together with the state of the art related with this tools.

The chapter begins by addressing what is probably the most important tool in this work: *the Clips expert system*. Since a detailed discussion of Clips is not a requirement of this work (plus there are specific manuals for the readers who wish to know every Clips detail), we analyze this tool with particular emphasis on the aspects of Clips against which we actually clashed while designing the *iClips* system. These elements will be used later to provide a better understanding of the design and the implementation sections.
The second argument described in the chapter is *the FACE project*.
Since the main purpose of the iClips system is an interaction tool for controlling a robotic FACE, it is appropriate to provide the reader with a brief description of the existing FACE project, its operations and uses.
Going further with the chapter, we introduce *the Kinect sensor*, a powerful device developed by Microsoft, and the Kinect SDK. We used these tools to implement some behaviour to test Clips on the FACE system. Obviously this does not imply that this particular sensor is fundamental or even the only one we could have used for the purpose, however it deserves a special place in the tools section essentially because it is not *just a sensor* (e.g. it is able to do more than a simple camera). In the Kinect section, we list the main features and advantages of this sensor.
At this point we focused on *the YARP Framework*, a standard tool for communication in robotics. The choice of YARP as the main communication protocol for the developed architecture is due specifically to its being a standard in this area.
The section describes the main features of YARP, its use, its contribution in a modular architecture (like ours), the tools associated with it (e.g. ACE, CMake, SWIG) and why we decided not to use this last tool (SWIG) as a YARP/`C#` wrapper, but to realize it by ourselves instead.
Finally, we introduce YARP Network and give a simple example of how it works.
The chapter then continues with an overview of *the .NET framework*, that is also faced from our point of view. That means we address it by exploring the features of the framework that fit better our goals, as we did with the Clips tool itself (the thesis does not want to be a treatise on the .NET framework

or Clips).
After a brief introduction on the .NET framework and the benefits it introduces, we focus primarily on its own concepts of reflection and custom attributes, giving examples of the latter.
The last tool described in chapter 2 is *the AvalonEdit editor*, a syntax driven editor we heavily used to build an interface allowing the user to have an Interactive Clips window as a viewport on the rules-based world.

Chapter 3 (ARCHITECTURE DESIGN) is the heart of this thesis.
It talks out the details of the system design, by explaining the reasons, the choices, the tools and how we actually used them, together with the issues and the challenges we faced. The chapter starts with an overview on the architecture design explaining how it can be seen as an *Hybrid deliberative/reactive* architecture. The related paradigm is then introduced and explained. The chapter goes on with a list of the requirements needed by the system, explaining in detail their characteristics.
At this point a complete description of the system is given, showing the architecture's structure and its components. A conceptual diagram is given to help the reader understanding. Then we see what are the building blocks of the architecture and how/where they are placed in it. The modular architecture concept is further described and it is specifically shown how these modules are built in terms of structure and functions/behaviour.
Another section of this chapter focuses on the iClips interface and how it is designed, what are its capabilities and how it can be used. The purpose of this section, in addition to the technical aspects, is to understand the importance in such an architecture of what we call a *"What you see what you get"* editor that encourages the declarative paradigm pervading the whole work.
The final part of this chapter focuses on aspects related to the YARP Module, which introduces a standard architecture for communication used in robotics and so it helps to have a distributed environment.

Chapter 4 (ARCHITECTURE IMPLEMENTATION) represents the most practical chapter of the whole work.

Basically, it can be viewed as related to the previous one, with the difference that in the design chapter we have described the tools, how they work, what they can do, the advantages they provide and how we used them, while in this particular chapter we show in practice, by using the code itself and small descriptions, the same concepts that have been already described in the previous chapter and that at this point should be clear to the reader.

In Chapter 5 (USE CASES), use cases are presented.
Use cases are mostly examples of how the system has been used together with proposals of how the system could be used. In fact, the architecture lends itself to a variety of purposes and can be used to implement/recreate several behaviours into the robot. The chapter proposes some use cases examples making the reader aware of what can be achieved through the system. The aim of this work is not to implement a specific behaviour into the robot but to provide the architecture with the means that would enable you to implement all the behaviour that you want.

The last chapter, **Chapter 6**, (CONCLUSIONS), presents the conclusions we have reached, and suggests the future developments which can be designed and developed starting from this architecture, together with the improvements which could be introduced.

Chapter 2

TOOLS and APPLICATION DOMAIN

This chapter is intended to people who want to familiarize with the tools and frameworks used in this work and with the existing projects it starts from. The chapter provides the reader with the necessary tools and concepts for a better understanding of what has been done and how.
The sections of this first chapter are named after the tools explained therein. However, it will not be an in-depth discussion (we refer to the related manuals and tutorials for this) but an overview on the main notions of each tool plus a more detailed discussion on the particular aspects this work mostly relies on.
People who already know these tools, or who are not interested in knowing them more deeply, can skip this chapter.

2.1 The CLIPS Expert System

One of the most important tools on which this thesis relies is Clips, an expert system tool developed by the NASA Space Center in the early 1980s, designed to facilitate the development of software which model human knowledge or expertise.
This section will give an idea of what an expert system is, and what it is

made for. The overview of the main points of Clips is then given by taking into account the key aspects of *iClips*.

An expert system can be defined as *"an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solutions"*. [JCG89]
This means that an expert system is a computer system that solves problems by emulating the decision-making ability of a human expert (a person who has expertise in a certain area).
The term *emulate* means that the expert system is intended to act in all respects like a human expert by using specialized skills or knowledge. Specialized knowledge is usually expected from specialized experts, because it is usually unknown or unavailable to most people. Anyway it might also be gathered from books, magazines, and general knowledge overall.

The '**C**' **L**anguage **I**ntegrated **P**roduction **S**ystem (Clips) is a powerful development and delivery expert system tool that provides a complete environment for the construction of rule-based expert systems. [Wyg89]

In the late 1950s, special programming languages that facilitate symbol manipulation were invented. Probably the most significant one is LISP (LISt Processing). Logic programming is a technology that comes fairly close to embodying the declarative ideal we manage to reach in this work: an ideal in which systems should be constructed by expressing knowledge in a formal language and problems should be solved by running inference processes on that knowledge.
Because of its simple elegance and flexibility, most AI research programs are written in LISP.
Clips, in fact, is a *LISP-based production system* which deals with rules and facts. It is written in C specifically to provide high portability, interoperability, speed, and low cost.

Other key features of Clips include a powerful rule syntax, an interactive development environment, high performance, extensibility, a verification/-

validation tool, extensive documentation, and source code availability. The ability to separate the development environment from the delivery environment (i.e. run-time modules) is a further strength of the tool.

The rule-based system used in Clips is implemented in a very efficient manner using the *Rete Algorithm*.

The **Rete Algorithm** is *an efficient pattern matching algorithm for implementing production rule systems.*[Fei82]
Rete is the Latin for "net". This algorithm handles in an elegant way one of the problem of major importance in expert systems with hundreds or thousands of rules: the efficiency.
In fact, no matter how good everything else is about a system, if a user has to wait a long time for a response, the system will be not used.
The Rete algorithm is a solution to this problem. It knows about all the rules and can apply any rule without having to try each one sequentially. Rete efficiency, indeed, is asymptotically independent of the number of rules and it is several times faster than any known alternative algorithm. The algorithm is a fast pattern matcher that obtains its speed by storing information about rules in a network. Instead of having to match facts against every rule on every recognize-act cycle, the Rete algorithm looks only for changes in matches on every cycle. Using Rete then, the computational cycle is based upon the propagation of differential changes in the working memory (that will be described later on). This greatly speeds up the matching of facts to antecedents (since the static data that don't change from cycle to cycle can be ignored) and reduces the computational complexity from that of a brute-force matching process in conflict set formation.
The Rete Algorithm is widely used and forms the computational basis for numerous commercial rule-based tools like Clips.

2.1.1 Expert Systems Building Blocks

We can now list and describe the components of an expert system. They basically consist of:

- *Working Memory.*
  The database of facts used by the rules. It represents the set of facts known about the domain. The elements in the working memory reflect the current state of the world.

- *Knowledge Base.*
  The rules on which the system relies. It represents the knowledge domain of the expert. Depending on the input data and the knowledge base, an expert system may come up with the correct answer, a good answer, a bad answer, or no answer at all.
  The general form of a rule is:

  If cond1 **and** cond2 **and** cond3 ...
  then action1, action2, ...

  The conditions cond1, cond2, cond3, etc., (also known as antecedents) are evaluated based on what is currently known about the problem being solved (i.e., the contents of the working memory). Each antecedent of a rule typically checks if the particular problem instance satisfies some condition. The consequents of a rule typically alter the working memory by incorporating the information obtained by the application of the rule. This could mean adding more elements to the working memory, modifying an existing working memory element or even deleting working memory elements. Consequents could also include actions such as reading input from a user, printing messages, accessing files, etc.
  When the consequents of a rule are executed, the rule is said to have been *fired*.

- *Inference Engine.*
  Probably the most important element of an expert system. It makes inference by deciding which rules are satisfied by facts or objects, prioritizes the satisfied rules, and executes the rule with the highest priority. It uses the information in the working memory in conjunction with the rules in the knowledge base to derive additional information about the problem being solved.

- *Agenda.*
  It's a prioritized list of rules created by the inference engine, whose patterns are satisfied by facts or objects in the working memory.
  In other words, it is the set of rules which can be fired at one particular moment. That is, if a rule has multiple patterns, then all of them must be simultaneously satisfied for it to be placed on the agenda. A rule whose patterns are all satisfied is said to be *activated*, but it has not yet been executed. Multiple activated rules may be on the agenda at the same time, in which case the inference engine must select one rule for *firing*.

- *User Interface.*
  The mechanism by which the user and the expert system communicate.

A simplified schema of an expert system is displayed in Figure 2.1.

At this point, it is clear that an expert system has a unique structure, different from traditional programs. In order to abstract this structure, we can see it as mostly divided into two parts, one fixed, independent of the expert system (the inference engine), and one variable (the knowledge base). The knowledge base and the working memory are the data structures which the system uses and the inference engine is the basic program which is used.
The basic execution cycle of the Clips Inference-Engine (IE) proceeds as follows:

1. If the rule-firing limit has been reached, execution is halted.

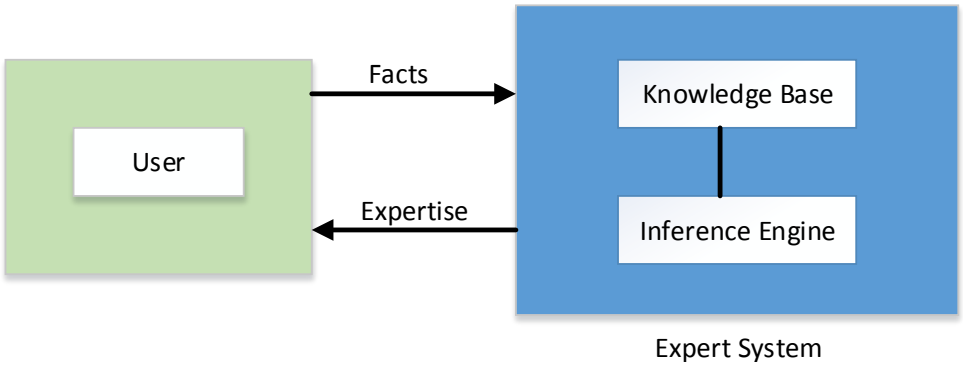

Figure 2.1: Basic Concept of an Expert System Function

2. If the rule-firing limit has not been reached, the top rule on the agenda (conflict set) is selected for execution. If there are no rules on the agenda, execution is halted. Otherwise, the right-hand side (consequent) actions of the selected rule are executed. The number of rules fired is incremented.

3. As a result of Step 2, rules may now be activated or deactivated. A new agenda is recomputed and the Clips IE returns to Step 1.

2.1.2 Expert Systems and AI

The connection between Expert Systems and Artificial Intelligence (AI) is very close. Expert Systems, indeed, together with robotics, vision, speech, natural language understanding and learning, human behaviour emulation and so on, are an important branch of Artificial Intelligence.
As a matter of fact, the scientific goal of AI is to understand intelligence by building computer programs (the AI agent) that exhibit intelligent behaviour and to implement functions which map perceptions to actions. An intelligent agent can be viewed as perceiving its environment through sensors and acting upon that environment through actuators:

"A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators. A robotic agent might have cameras and infra-red range finders for sensors and various motors for actuators. A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets". [SR09]

Artificial Intelligence is concerned with the concepts and methods of symbolic inference, or reasoning, by a computer, and how the knowledge used to make those inferences will be represented inside the machine. One of the fundamental working hypothesis of AI is in fact that intelligent behaviour can be precisely described as symbol manipulation and can be modeled with the symbol processing capabilities of the computer. The AI programs we are going to use are knowledge-based or expert systems. They achieve expert-level competence in solving problems belonging to specific areas thanks to domain-specific knowledge.
Lastly, the process of building an expert system is known as *knowledge engineering*. This process is an advantage itself since the knowledge of human experts must be put into an explicit form in order to become an input for the computer. Due to this reason, it can be examined for correctness, consistency, and completeness. The knowledge may then have to be adjusted or re-examined, which further improves the quality of the knowledge itself. The knowledge concept is then strictly connected to the concept of *cognition*. Cognition is the study of how humans process information, and this is very important if we want to make computers to emulate human experts. Now, since one of the major roots of expert systems is the area of human information precessing called **cognitive science**, it's obvious why expert systems have turned out to be so important to Artificial Intelligence.

2.1.3 CLIPS Specifications

The goal of this section is to supply the reader with both the main theoretical and technical concepts of Clips, in order to give him/her an idea of the noteworthy aspects of the *iClips* tool itself.
Clips is a multi-paradigm programming language that provides support for rule-based, object-oriented, and procedural programming. During these years, Clips has undergone continual refinement and improvement and it is now used by thousands of people around the world. Like other expert system languages, Clips deals with rules and facts, and it is used in business, science, engineering, manufacturing and many other fields.

2.1.3.1 Basic Elements

There are three ways to represent information in Clips:

- *Facts.*
  To solve a problem, a Clips program must have data or information with which it can reason. A "piece" of information in Clips is called a fact. Facts consist of a relation name followed by zero or more slots and their associated values. The order in which slots are specified is irrelevant.
  It is possible to assert a fact through the following command:

  `(assert <fact>+)` (i.e. `(assert (weather rainy yes))`)

- *Objects.*
  Objects in Clips are split into two important categories: primitive types and instances of user-defined classes. These two types of objects differ in the way they are referenced, created and deleted as well as how their properties are specified.

- *Global Variables.*
  The *defglobal* construct allows variables to be defined which are global

in scope throughout the Clips environment. That is, a global variable can be accessed anywhere in the Clips environment and retains its value independent of other constructs.

And three ways to represent knowledge:

- *Rules.*
  Which are primarily intended for heuristic knowledge based on experience. Rules can be typed directly into Clips or loaded in from a file of rules. The general format of a rule is:

```
(defrule <rule name> [<comment>]
<conditional-element>* ; Left--Hand Side of the rule
=>
<actions>* ); Right--Hand Side of the rule
```

  i.e.

```
(defrule example-rule "Example of a simple rule"
(weather rainy yes)
=>
(assert (umbrella needed)))
```

- *Deffunctions.*
  Which are primarily intended for procedural knowledge.

- *Object-oriented programming.*

Clips supports three programming paradigms: Forward chaining (rule-based), Object Oriented, and Procedural. We will focus on the first one.
Facts are one of the basic high-level forms for representing information in a Clips system. [Gia02]
Each fact represents a piece of information which has been placed in the current list of facts, called the **fact-list**. Furthermore, facts are the fundamental unit of data used by the rules. Facts may be added to the fact-list (using the **assert** command), removed (using **retract**), modified or duplicated. Whenever a fact is added it is given a unique integer called the **fact-index.**

Clips provides eight primitive data types for representing information. These types are **float, integer, symbol, string, external-address, fact-address, instance-name** and **instance-address** [Gia06b]. Numeric information can be represented using float and integers. Symbolic information can be represented using symbols and strings.

The three predominant Clips constructs are: `deftemplate` for defining working memory structures or templates, `defrule` for defining productions, and `deffacts` for defining facts, i.e., creating instances of templates.

2.1.3.2 Deftemplates

Since the `deftemplate` construct plays an important role in this thesis, it is worthwhile to be further described.

The keyword `deftemplate` stands for *"define template"* and is similar to a record structure definition. That means, through the `deftemplate` construct, it is possible to define a group of related fields in a pattern similar to the way in which record is a group of related data. A `deftemplate` is a list of named fields called slots, and it can help you in asserting facts whose pattern have a well-defined structure.
A slot can be of two types: single-slot or a multi-slot. Any number of single or multi-slot slots may be used in a `deftemplate`. The construct allows access by name, so the order of the fields does not matter. Writing a slot is very easy: you just need to give the field name (attribute) followed by the field value. In general, a `deftemplate` with N slots has the following general structure:

```
(deftemplate <relation--name> [<optional--comment>]
(slot-1)
(slot-2)
(...)
```

```
(slot-N)
```

An example will make it more clear.
Consider the following `deftemplate "status"` relation, which can be summarized by the following table:

Attribute	Value
name	”Timmy Tommy”
occupation	student
age	23

You can set the values in many ways: by explicitly giving them, by listing them, by giving a range of values or as a list of allowed values.
The `deftemplate` for the ”status” relation could be:

```
(deftemplate status "personal information"
(slot name (type STRING) (default ?DERIVE))
(slot occupation (type SYMBOL) (default unemployed))
(slot age (type NUMBER) (default 30)))
```

So we have the following components: a `deftemplate` relation name, attributes called fields, the field type -which can be any one of the allowed types (`SYMBOL, STRING, NUMBER,` and others)-, the default for the field value. This particular `deftemplate` has three single-slot slots called name, occupation, and age. The `?DERIVE` keyword selects the appropriate type of constraint for that slot, e.g., the null string , `""`, for a slot of type `STRING`.
Then, by `(assert (prospect))` we can assert a fact that is built as the previous template.
The `deftemplate` default values are inserted by Clips if no explicit values are defined. In the `deftemplate`, `NUMBER` is not a primitive field type like `symbol, string, integer`, and `float`: it is a compound type that can be `integer` or `float`, for cases in which the user does not care what type of numbers are stored.

The construct turns out to be very useful in our architecture both to the iClips and YARP tool. We will see exactly how in the next chapter.

2.1.3.3 Conflict Resolution

One of the most important aspects which makes Clips perfectly suited to our purposes is the possibility to assign priority to the rules. The **salience rule property** allows the user to assign a priority to a rule: let's see how this works.
The starting point is that rule-based programming paradigm implemented by a production system differs significantly from (conventional) imperative programming. In the rule-based paradigm, there is no sequence of commands to be executed. Instead, a sequence of production firings takes the system from an initial state to a final state. The order of the productions used is not explicitly specified a priori, because the antecedents or condition elements of the production must be satisfied before it can be considered for firing.
However, the main aspects of the production system computation are controllable in several ways:

1. Through the control of rule salience (the one we were mentioning before);

2. Through the design of the productions themselves;

3. Through the choice of conflict resolution strategies used.

The Clips conflict set is the above cited `agenda`, that is the list of rules ranked in descending order of firing preference, called *activated* rules. When changes to the working memory cause a rule to no longer be in the conflict set, it is said to be *deactivated*.
Placement of rules on the agenda is then determined by *salience*, the aforementioned property. This feature allows the specification of the relative importance of rules independently from any conflict resolution scheme. Salience may be dynamically controlled. When multiple rules are in the agenda, the rule with the highest priority/salience will fire first: that means placement of rules on the agenda is determined by this very property. The salience value

is an expression evaluated as an integer in the range from -10000 to +10000 (the default value is zero).
When a rule is newly activated, its placement on the agenda is based (in order) on the following factors:

1. Newly activated rules are placed above all rules of lower salience and below all rules of higher salience.

2. Among rules of equal salience, the current conflict resolution strategy is used to determine the placement among the other rules of equal salience.

3. If a rule is activated (along with several other rules) by the same assertion or retraction of a fact, and the preceding two test are unable to specify an ordering, then the rule is arbitrarily (not randomly) ordered in relation to the other rules with which it was activated.

Clips provides seven user-selectable conflict resolution strategies: depth, breadth, simplicity, complexity, lex, mea, and random.

- *Depth.*
  This strategy ranks productions based upon the recency of rule activation.

- *Breadth.*
  It is implemented by placing the newest activations of rules on the bottom of the agenda.

- *Simplicity/Complexity.*
  This agenda ranking is based upon specificity or generality of a production, measured in terms of antecedents comparisons in the rules. For simplicity, the more comparisons, the lower the rule is on the agenda. For complexity, ranking is reversed.

- *Lex.*
  This strategy emphasizes recency of the antecedents.

- *Mea.*
  It forms the agenda by sorting activated productions using their time tag.

- *Random.*
  As the name suggests, it assigns a random number to each production to determine the placement of it on the agenda.

The default conflict resolution strategy is depth. The default **depth** conflict resolution strategy implements a depth-first search by placing newly activated rule instantiations at the top of the agenda. If a new fact activates more than one instantiation of a rule at a time, the order of those instantiations is arbitrary.

A strategy is selected using the `set-strategy` command, with syntax:

```
(set-strategy <strategy>)
```

When the conflict resolution strategy is changed, the agenda is reordered. [Sch09]

2.1.3.4 Integration with other languages

When using an expert system, two kind of integration are important: embedding Clips in other system, and calling external functions from Clips.
Using Clips as an embedded application allows the easy integration of Clips with existing systems. This is useful whenever the expert system is a small part of a larger task or needs to share data with other functions. In these situations, Clips can be called as a subroutine and information may be passed to and from Clips.

As we ourselves have experienced with iClips, a realistic production system may need to interface with other applications.
As shown in the Advanced Programming Guide [Gia06a], Clips supports integration in two ways:

1. Clips may be embedded in user code for another application; and

2. Clips may call external (user-written) applications.

When running Clips as an embedded program, many of the capabilities available in the interactive interface are available through function calls.
In this work, the Clips engine has been embedded through the *CLIPSNet* wrapper, a .NET library for embedding Clips into .NET applications that will be better detailed later on, when going deep into the iClips project.

2.1.3.5 Routers

Routers are the mechanism used by Clips to handle I/O streams by specifying different sources and destinations. Since Clips has been designed to be embedded in other applications, Router is the way to do stream redirection. It is possible to define a new router by defining it through a Clips environment, a logical name, and a priority. In Clips, logical names are used to send I/O requests without having to know which device and/or function is handling the request. When Clips receives an I/O request, it begins to query each router to discover whether it can handle an I/O request. Routers with high priorities are tried before routers with low priorities. Priorities are very important when dealing with multiple routers that can process the same I/O request.

Our new router in iClips will *intercept* all the I/O requests from Clips.
To define a new router, we had to override its main functions. These are:

- *Print.* The print function associated with that router.
- *Query.* This function is in charge to intercept any I/O requests that the standard interface would handle. In addition, it also handles requests for the logical name top.
- *Getc.* The get character function associated with that router.
- *Exit.* It calls the exit function associated with each active router before exiting Clips.

2.1.4 The CLIPSNet Library

As we already know from the previous chapters, the main goal of this work is the introduction of the expert system Clips within the existing FACE project. Since we deal with a .NET application, we need a wrapper allowing Clips to be embedded in our system. This possibility is offered for free by the *CLIPSNet Library*. [Soua]
The CLIPSNet Library is a .NET library developed in March 2008 for embedding Clips in to .NET applications. It allows a convenient and easy integration of the Clips expert system in .NET projects, and due to this it perfectly fits in our architecture.

Through the CLIPSNet library, we can access the same advantages and benefits of the Clips expert system itself. When running Clips as an embedded program in fact, many of the capabilities available in the Clips syntax are available through function calls.
Despite the poor (almost absent) documentation, the CLIPSNet library turns out to be an easy tool to use: by using a very intuitive syntax, it makes the Clips structures and functions available in .NET. However these structures and functions are called and used in a slightly different way. That means, through the CLIPSNet Library the model by which we interact with Clips is a little bit different from the one described in the Clips manual and tutorials. For example, the assert syntax is as follows:

CLIPS syntax:

```
(assert (color green))
```

CLIPSNet syntax:

```
clipsEnv.AssertString(string.Format("(color green)"));
```

In order to have an overview of the library and see how it is organized internally, a simplified class diagram has been generated and is available on [Figure 2.2].

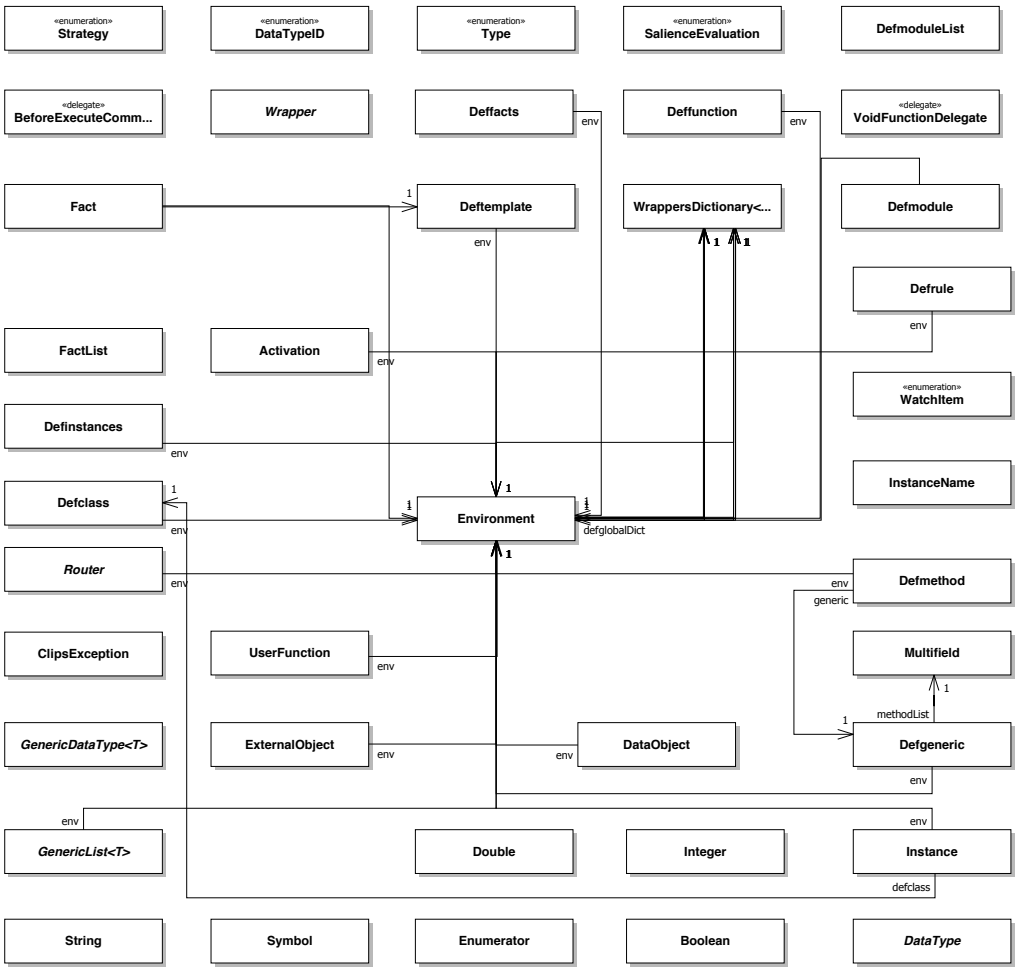

Figure 2.2: A CLIPSNet simplified Class Diagram

The use of the CLIPSNet library implies a more detailed study of some noteworthy aspects which need to be discussed further. The first one is the data format.

Even though Clips may be embedded within a .NET program through the CLIPSNet library, the integration of Clips with the .NET environment requires an understanding of which data format is used by Clips and the .NET framework to exchange parameters. To improve usability and to minimize the amount of recoding needed for each module, we must ensure argument

compatibility.
Passing arguments from Clips to an external function and from external functions to Clips is generally a possible operation when embedding the Clips expert system. This operation makes .NET functions usable from Clips.
In such a situation however the user must ensure arguments compatibility by using necessarily the `Clipsnet.Datatype` format.
The `Clipsnet.Datatype` namespace contains nine data types, listed as follows: `Boolean, Double, ExternalObject, GenericDataType<T>, InstanceName, Integer, Multifield, String,` and `Symbol`.
The use of these data types represents the only requirement-constraint to properly interact with Clips from the .NET environment: the .NET functions that we expose to Clips must accept and return Clips data types as parameters and return values, respectively. Once you've done so, you can assert facts from `C#` in the rules world and from this last one you can call `C#` methods that have an appropriate signature.

The general way to expose a function to Clips is by using the *delegate* declaration. We can see how it works through an example.

Let's say that we have a `turnFace` function in `C#`, and we want to expose it in Clips in order to make it accessible and usable from the rules world (more specifically, in order to make it usable from the `.clp` file, the Clips file containing the specific module rules).
In the .NET world, we have the following declarations:

```
1 delegate void turnFaceDelegate(CLIPSNet.DataType o);
2 var v = new CLIPSNet.UserFunction(env,
3           new turnFaceDelegate(turnFace), "turnFaceFun");
```

This second statement links the `C# turnFace` function to the `turnFaceFun` Clips function. Now we can use it directly from Clips.
Here it is a snapshot of the `.clp` file using it.

```
(defrule MAIN::TurnFace
       ?factNum <- (turningFace ?howmuch ?where)
       =>
       (bind ?angle (computeAngle ?howmuch ?where))
       (if (neq ?angle 0) then (turnFaceFun ?angle))
       (retract ?factNum))
```

This example should allow the reader to understand the basic concepts needed for correctly exposing functions to Clips.
However, as shown by the example, the Clips-.NET integration requires each single function to have its own delegate written specifically for it. This causes the code to be unclean and, above all, hard to maintain. Due to this we designed a generic delegate which makes `C#` functions usable from Clips through *annotations*. We will further discuss this aspect in section 4.1 when delving into the iClips module.

2.2 The FACE Project

FACE (**F**acial **A**utomation for **C**onveying **E**motions) is a humanoid robot developed at the Interdepartmental Research Center *"E. Piaggio"* of the University of Pisa in collaboration with the IRCCS Stella Maris Institute of Calambrone (Pisa).

The human robot is capable of expressing and conveying emotions and empathy. The project aims to enable autistic subjects to improve the way they deal with emotional and expressive information, through familiarity and contextual information presented in a stepwise way. Subjects' behaviour and responses are, as a matter of fact, monitored using sensors and then processed and fed back to the android to modulate and modify its expressions. [MBL+10]
FACE consists of a passive articulated body equipped with a believable facial display system based on biomimetic engineering principles. The latest prototype of the FACE head is a female face built by means of life-casting techniques. Aesthetically it represents a copy of a real head.

Figure 2.3: The FACE Android

The artificial skull of FACE is covered by a special skin made of a particular material (a silicone elastomer that contains up to 70% air by volume, patented by HansonRobotics) that allows the skin of the robot to look and move like human skin.
The realism is improved by the actuating system of the robot, a system controlled by a SSC-32 serial servo controller with 32 servo motors (Figure 2.5), of which 5 neck servos allow pitch, roll and yaw movements of the head, and the others allow facial expressions. The motor cables act as tendons moving FACE skin and allow human facial expressions to be re-created (Figure 2.3).

System modules are integrated using the Robotics4.NET framework, a programming framework which aims to support the development of control software for robotic systems. [ACCE05]
FACE can express and modulate the six basic *"Ekman"* emotions (happiness, sadness, surprise, anger, disgust, fear) in a repeatable and flexible way thanks to its artificial muscular architecture. This six basic emotions, due to their simple and stereotypical nature, are easily accepted by autistic patients

Figure 2.4: Robot FACE expressing the six basic Ekman emotions

(Figure 2.4).

The android is then used to engage the child in simple interactions based on exchanging and learning emotions through the imitation of the android's facial expression and behaviours.[MLZ+12]
The environment is then perceived through a number of different sensors, in part mounted on the android (like a CCD camera in the right eye used for the face tracking of the subject), in part on the surrounding environment (Microsoft Kinect and others) and in part on the patient.

One of the most important contributions of this thesis derives from an idea proposed in the *Future Works* section in a previous work [Laz09].
In fact, when starting this work, the behaviour of the robot was purely reac-

tive in response to external stimuli. For instance, if the patient is moving, the robot eye-camera can be used to track his/her face and to turn the robot's neck and eyes in order to keep the subject in view as a real person would do. The FACE project and its software framework were able to control the servo motors to define the facial expressions of the android.
Then, as next step, it was proposed to modulate the social behaviour of the robot according to a self-adaptive control algorithm which could enable FACE to choose autonomously its expressions and even its behaviour according to its *"mood state"*. This feature could make the robot act more like a human, having its own mood and feelings. This would improve the likelihood of the child developing appropriate mechanisms to infer the robot's "intentions" or emotional state.

Hence the main goal of the *iClips* system stems from an actual need. The way that has been chosen to meet this need is the already mentioned expert system: Clips.

2.3 The KINECT Sensor

One of the most powerful device used in this work is the Kinect sensor.
It is highly noteworthy because it's not *"just a sensor"*. From our point of view, it is instead a particular sensor which enables advanced gesture recognition, facial recognition and voice recognition. We decided to use this specific sensor as a starting point both to familiarize with the architecture and to determine how to generalize it. In fact, once we generalized the architecture, we realized that Kinect sensor represents only one sub-case of what could be with any other sensor.

Kinect is a motion sensing input device by Microsoft created in 2010, at first for the Xbox 360 video game console, then also for Windows PCs. The Kinect depth sensor, the camera combination and some of its software were created by PrimeSense, a 3D sensing and Natural Interaction solutions company that is well known for licensing the hardware design and chip used in

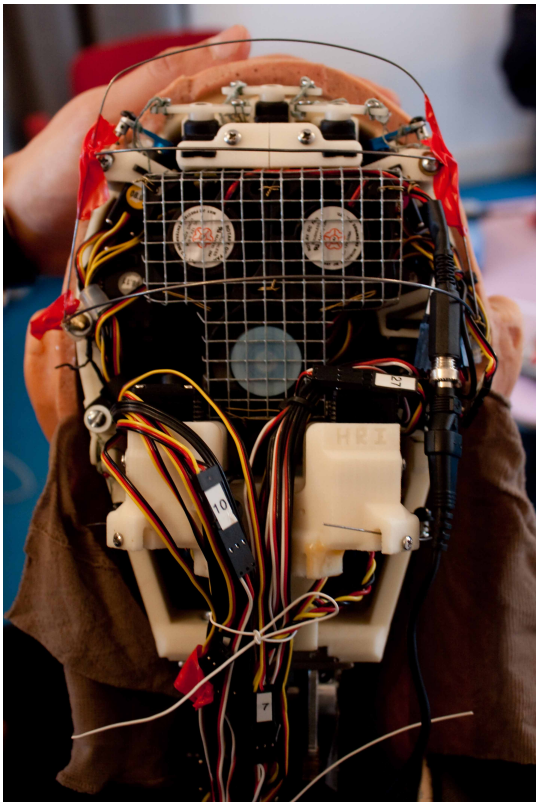

Figure 2.5: Robot FACE

Microsoft Kinect motion-sensing system. Microsoft Research developed the software that is used to handle the facial and speech recognition as well as the skeleton tracking.
In the Xbox 360 the Kinect enables users to control and interact with the console without the need to touch the game controller, through a natural user interface using gestures and spoken commands. [Micb]
On February 2012, Microsoft announced the release of a non-commercial Kinect **S**oftware **D**evelopment **K**it (SDK) for Windows.
This led many developers to research possible applications of Kinect that go beyond the system's intended purpose of playing games: Microsoft Kinect SDK meets this need by providing access to much of the functionality available to Xbox developers and by offering the potential to transform how people interact with computers and Windows-embedded devices in multiple fields.

The SDK provides the tools and APIs, both native and managed, needed to develop Kinect-enabled applications for Microsoft Windows using managed or unmanaged code in a .NET environment.
Developing Kinect-enabled applications is essentially the same as developing other Windows applications. In addition to the usual redistibutable libraries, the applications need to be deployed with Windows 7 compatible PC drivers for the Kinect device (included in the Kinect SDK). The SDK provides support for the features of the Kinect, including color images, depth images, audio input, and skeletal data.
The SDK also includes the following features:

- *Raw sensor streams:*
  Access to low-level streams from the depth sensor, color camera sensor, and four-element microphone array.

- *Skeletal tracking:*
  The capability to track the skeleton image of one or two people moving within the Kinect field of view for gesture-driven applications.

- *Advanced audio capabilities:*
  Audio processing capabilities include sophisticated acoustic noise suppression and echo cancellation, beam formation to identify the current sound source, and integration with the Windows speech recognition API.

- *Sample code and Documentation.*

2.3.1 Kinect Specifications

The Kinect sensor is a horizontal bar connected to a small base with a motorized pivot. It features an RGB camera, a depth sensor and a multi-array microphone, which provide full-body 3D motion capture, facial recognition and voice recognition capabilities (Figure 2.6).
The depth sensor consists of an infra-red laser projector combined with a

Figure 2.6: The Kinect Sensor

monochrome CMOS sensor, which allows the unit to output three-dimensional position data in real time and captures video data in 3D under any ambient light conditions.

The sensing range of the depth sensor is adjustable, and the Kinect software is capable of automatically calibrating the sensor based on the physical environment, accommodating for the presence of furniture or other obstacles. The Kinect sensor has a practical distance range limit of 1.2 3.5 m. The sensor has an angular field of view of 57° horizontally and 43° vertically, while the motorized pivot is capable of tilting the sensor up to 27° either up or down.

The Kinect data is organized as a stream of two 640 x 480 images acquired at a rate of 30 fps (frames per second). One of these is an ordinary 24-bit RGB video image. The other is an 11-bit depth image from which (x, y, z) positional data may be computed.

Inferring the body position is a two-stage process: first compute a depth map p (using structured light), then infer body position (using machine learning techniques). According to the information supplied to retailers, Kinect is capable of tracking up to six people at the same time, including two active players for motion analysis with a feature extraction of 20 joints per player. [Micb]

2.4 The YARP Framework

YARP, **Y**et **A**nother **R**obot **P**latform, is an open source library designed to support software development for humanoid robotics.
The goal of YARP is to minimize the effort devoted to infrastructure-level software development by facilitating code reuse, modularity, to maximize research-level development, collaboration and code-sharing, both in terms of space and time. [MFN05]

In the *iClips* system, YARP is one of the most important requirement set up from the beginning.

There are different reasons why YARP is so important to humanoid robotics. Humanoid robotics is a field of research, with constant flux in sensors, actuators, and processors that stretches the limits of current technology: in such a setting, one processor in never enough.
Furthermore, humanoid robotics is notoriously hardware-specific and task-specific. But hardware and target tasks change frequently, even within the lifetime of one project. Our humanoid robots are complex to build and maintain; this is the reason why they need to be *modular*.
The most practical way to handle this challenge is to have a robot control system as a set of processes running on a set of computers: code, indeed, is easier to maintain and reuse if it is organized in small processes, each one performing a simple task. [Figure 2.7]
There are several frameworks for modular robot systems, YARP is one of them. [FMN07]
It is a software platform that eases the above mentioned tasks and improves the software quality on robot platforms. For projects of a reasonable size YARP encourages a modular approach and makes it easy to write processes that are location-independent and that can run on different machines without code changes. This means that software is ideally divided in independent components, that can be developed and maintained by different people having distinct competences.

Figure 2.7: The YARP modular approach. We assume a set of processors (which may be on the robot or elsewhere) and diversity (different devices, operating systems, processors, languages, libraries, etc).

Moreover, a modular software platform is flexible, that means obsolete modules can be then removed and replaced with newer ones without catastrophic effects. As long as enough resources are available, the addition of new components should cause minimal interferences with the overall behaviour of existing ones. This is fundamental to make the software stable and long-lasting, without compromising our ability to constantly change sensors, actuators, processors, and networks.
In robotics, in fact, dependencies between modules need to be minimized also from the point of view of run-time performance.
By following a modular approach, we would like our software to be as flexible as possible and adaptable to the needs of users and to the platforms that they work on. In YARP, unavoidable dependencies have been made as localized as possible to modules that can be compiled or not depending on the underlying system and user choices.

Since we choose a modular approach ourselves, YARP is naturally part of the system infrastructure this thesis relies on.

But *modularity* alone is not a solution to software reuse, since different architectures or frameworks may be mutually incompatible. It's important that the developed modules can fit together. The main features of YARP include support for inter-process communication across different software and hardware platforms.

YARP is written almost entirely in `C++`. In robotics this is a good choice because `C++` allows writing very efficient code and interfacing with the hardware at the lowest level. `C++` is portable and widely supported, but the compilation process varies a lot depending on the platform and development environment.
The **CMake** tool (**C**ross **P**latform **M**ake [Cma]) overcomes this drawback by making projects easy to compile in a wide range of integrated development environments. CMake is a cross-platform, open-source build system used to control the software compilation process using simple configuration files which are independent both from the platform and the compiler. It produces build files for the environment of choice, starting from a language independent description. Through CMake the build process of YARP is robust, simple and flexible.

Thanks to CMake, the project is not restricted to just one particular development environment. But the code itself may still have operating-system dependencies that can prevent its portability.
YARP decided to reduce the dependencies with the operating system by using **ACE** (**A**daptive **C**ommunication **E**nvironment [Cse]), an open source library that provides an excellent portable interface to a wide range of operating systems, dealing with the details so you don't have to. YARP uses ACE in its implementation, but doesn't require YARP users to do so.

Another free and open-source tool is **SWIG** (**S**implified **W**rapper and **I**nterface **G**enerator [Swi]), that allows *Language Portability* and makes YARP easy to use from many different languages. SWIG is an automatic wrapper tool that creates an interface to YARP for you in the language of your choice. It takes `C/C++` source code and generates "*wrappers*" for it, usable from many different languages.

We tried to use SWIG, but we were unsuccessful.
Briefly, the reason lies in the fact that we faced non-trivial data structures, as opposed to what the automatic wrapper expected to handle. We then needed an ad-hoc marshalling process since the automatic one was just not possible, and we decided for Managed `C++`.
A proper section will face this problem when we'll describe the project in detail.

2.4.1 The YARP Network

YARP supports building a robot control system as a "collection of programs" communicating in a *peer-to-peer* way, with a family of connection types that can be swapped in and out to match our needs. These connection types are named *carriers* and they corresponds roughly to the "transport" protocol used to carry data.

Communication takes place through these *connections* between named entities called *ports*. These form a directed graph, the "YARP Network", where ports are nodes, and connections are the edges. [Figure 2.8]
Each port is assigned a unique symbolic name and is registered by this with a "name server": this is the first operation each port must perform.
The goal is to ensure that if you know the name of a port, that is all you need in order to be able to communicate with it from any machine. When the user is done with the port, it can be stopped, unregistered, and eventually destroyed.
The name server is a YARP program that maintains a list of all YARP ports

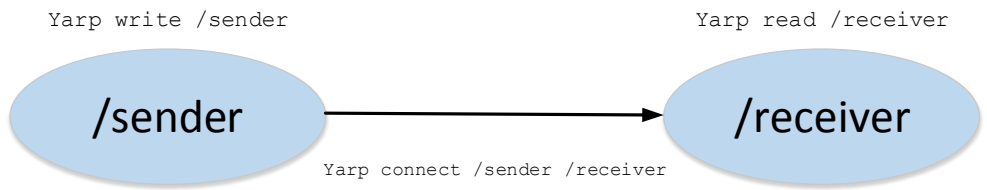

Figure 2.8: YARP Network Example

and how to connect to them by recording their contact information in a configuration file (the `yarp.conf` file). The name server maps symbolic names (strings) into a triplet composed of IP address, port number, and interface name. It has a YARP port itself usually named "`/root`". All other YARP programs communicate with the name server through this port.
The YARP network could be defined as a *Route Reflector* model [IET]: we can imagine it as kind of a star-shaped model, that means a distributed model but with a single point of failure identified with the YARP server.

The YARP network interface provides methods for manipulating parts of the network, such as creating or removing connections and ports.

Communication in YARP follows the *Observer pattern* [Proa]. The state of port objects can be delivered to any number of observers, in any number of processes distributed across any number of machines. YARP manages these connections in a way that insulates the observed from the observer and, just as importantly, insulates observers from each other.
In YARP, a port is an active object which sends/receives data to/from any number of other ports. Connections between ports can be freely added or removed (either programmatically or at run-time), and may use different underlying transports (e.g. TCP, UDP, multicast, shared memory). Communication is fully asynchronous, hence messages are not guaranteed to be delivered unless special provisions are made.
An example of a YARP application could be the acquisition of images and their delivery to many machines performing the processing in parallel. [Fig-

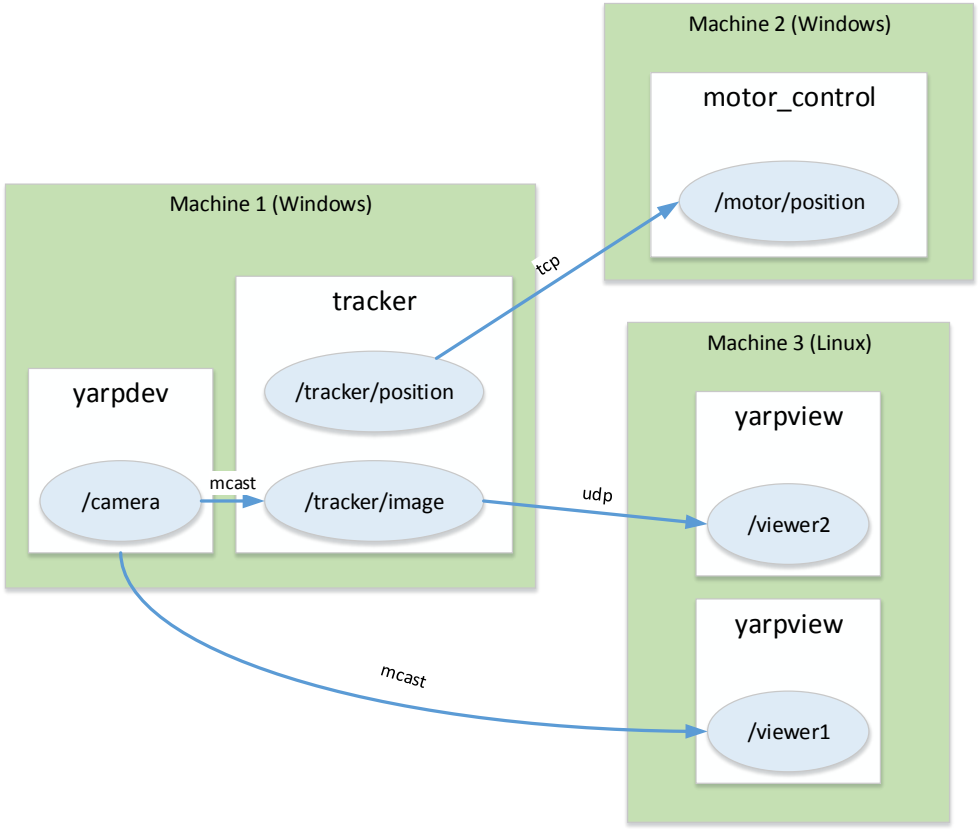

Figure 2.9: A Network Example of Ports

ure 2.9]

Furthermore, an example of a simple YARP sender and receiver code is given in Listing 2.1 and Listing 2.2.

Listing 2.1: YARP Sender Example

```
1 #include<yarp/os/all.h>
2 #include<stdio.h>
3 using namespace yarp::os;
4
5 int main() {
6     Network yarp;
7     Port output;
8     output.open("/sender");
```

```
    int top = 100;
    for (int i=1; i<=top; i++) {
        // prepare a message
        Bottle bot;
        bot.addString("testing");
        bot.addInt(i);
        bot.addString("of");
        bot.addInt(top);
        // send the message
        output.write(bot);
        // wait a while
        Time::delay(1);
    }
    output.close();
    return 0;
}
```

Listing 2.2: YARP Receiver Example

```
#include<yarp/os/all.h>
#include <stdio.h>
using namespace yarp::os;

int main() {
    Network yarp;
    Bottle bot;
    Port input;
    input.open("/receiver");
    Network::connect("/sender","/receiver");
        while(input.read(bot))
        {
                printf("Got message: %s\n", bot.toString().c_str());
        }
    input.close();
    return 0;
}
```

This short example shows how to set up a simple YARP Network for exchanging integers.

Ports for sending/receiving integers are created both in the sender and in the receiver, then a statement instructs the ports to register with the name

server with the arbitrary name "/sender" and "/receiver", respectively.
We can now connect the two processes by using the `Network::connect("/-sender","/receiver")` command. The next step is to wait for data from the input port and send data through the output port.
The connection protocol is the protocol used for single connection from an output port to an input port.
It has two main phases, the handshake phase and the message phase. We begin once the sender has successfully opened a bidirectional streaming connection of some kind to the receiver. The communication starts with the handshake phase:

- Transmission of protocol specifier: Sender transmits 8 bytes that identify the "carrier" that will be used for the connection.
- Transmission of sender name.
- Transmission of extra header material.

At the end of this phase, both the sender and the receiver are aware of the connection being established (carrier and port). Then the message phase (transmissions of index and payloads) takes place.

YARP networks (and ports) can deal with any data type. If you communicate between machines with different operating systems and compilers, you may need to be careful if you send your own custom data-type, but apart from that there are no limitations. We ourselves faced this problem in *iClips*.
The YARP communications system provides a standard data format for simple data types. [FMN12] This format is called the "*bottle*" and is a potentially nested list of some primitive types like integers, floating-point numbers, and strings, with a well defined representation in binary and text form. The name of this class comes from the idea of "throwing a message in a bottle" into the network and hoping it will be eventually picked up.

Complex data types are dealt by specializing the port `C++` template for the new complex data type and providing serialization and deserialization

functions. Serialization is done by providing lists of memory blocks, to minimize copies. As we briefly mentioned already, support for marshalling is not built into the library. Ports are implemented as C++ templates and specialized to the type of the data to be transmitted or received. This creates a very clean and consistent client interface.

Let's see how complex data are dealt through the following example. Suppose we are handling this structure:

```
1 class Target {
2 public:
3         int x;
4         int y;
5 };
```

In this case, the Bottle class is not useful to us. Instead, it is possible to create a buffered port for it this way:

`BufferedPort<BinPortable<Target>>` port.

The `yarp::os::BinPortable` class tells YARP that you want to send the Target type across the network by encoding it exactly as it is represented in memory.

The method `yarp::os::BinPortable::content()` will give you access to the actual Target object. If you want to use it between machines with different compilers, it is better to define the type using NetInt32 instead of int, to ensure that the types are compatible. Usually, rather than sending memory images across the network, it is better to provide explicit serialization methods for your class by implementing the `yarp::os::Portable` interface. Our class becomes:

```
class Target : public Portable {
public:
        int x;
        int y;
        virtual bool write(ConnectionWriter& connection) {
                connection.appendInt(x);
                connection.appendInt(y);
                return true;
        }
        virtual bool read(ConnectionReader& connection) {
                x = connection.expectInt();
                y = connection.expectInt();
                return !connection.isError();
        }
};
```

These take care of using neutral formats for your data types. Now you no longer need the BinPortable wrapper. It is possible to use:
`BufferedPort<Target>` port;

YARP main components can be broken down into:

- `libYARP_OS`: for interfacing with the operating system(s) to support easy streaming of data across many threads across many machines.
- `libYARP_sig`: for performing common signal processing tasks in an open manner easily interfaced with other commonly used libraries, for example OpenCV.
- `libYARP_dev`: for interfacing with common devices used in robotics.

These components are maintained separately. The core component is `libYARP_OS`.

The main command-line commands, which are useful to perform most common operations in the YARP network are: `check`, `clean`, `conf`, `connect`, `detect`, `disconnect`, `exists`, `help`, `name`, `namespace`, `read`, `run`, `server`, `terminate`, `wait`, `where`, `write`.

2.5 The .NET Framework

The .NET Framework, developed by Microsoft in the late 1990s, is a software framework usually running on Microsoft Windows platforms. [Mica]
The languages used in this work, primarily `C#` and `C++`, are strongly supported from this framework. The IDE that was used is the Microsoft Visual Studio IDE, an integrated development environment primarily used for .NET software. It provides language interoperability across several programming languages like `VB, C#, JScript, VBScript` etc. and it includes a large library that is available to all these supported programming languages. This means each language can use code written in other languages.

Programs written for the .NET Framework execute in a software environment, known as **C**ommon **L**anguage **R**untime (**CLR**), an application virtual machine that runs the code and provides important services such as security, code execution, memory management, garbage collection and exception handling which make the development process easier.
The class library and the CLR together with other blocks constitute the .NET Framework. [Figure 2.10]
Common Language Runtime also provides an abstraction layer over the operating system. The .NET Framework has many interesting design features and components:

- *Interoperability.*
  This is one of the features our project heavily relied on. Due to the different languages, systems and wrappers we used, interoperability was a mandatory feature to provide. It is not difficult to explain why we need it.
  Since computer systems commonly require interaction between newer and older applications, the .NET Framework provides means to access functionality implemented in newer and older programs executing outside the .NET environment. Access to COM components is provided in the `System.Runtime.InteropServices` and `System.Enterprise-`

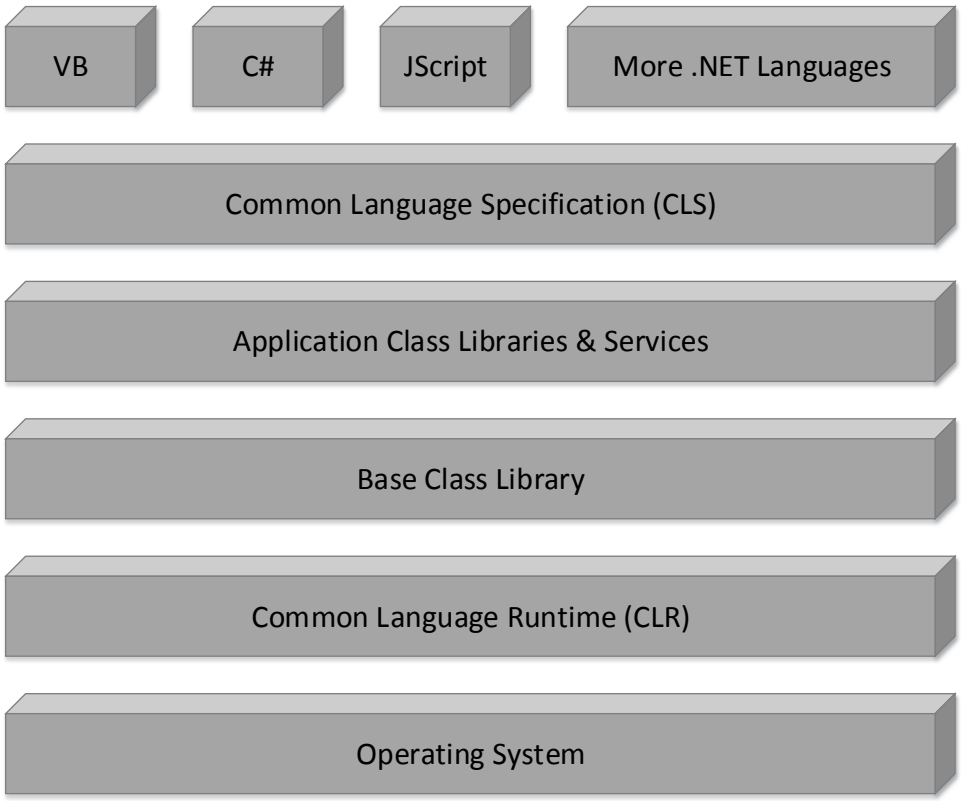

Figure 2.10: The .NET Framework

`Services` namespaces of the framework; access to other functionalities is achieved using the P/Invoke feature.

- *Common Language Runtime engine.*
  The Common Language Runtime serves as the execution engine of the .NET Framework. All the .NET programs execute under the supervision of the CLR, which guarantees certain properties and behaviours in the areas of memory management, security, and exception handling. The two main components of Common Language Runtime are the **C**ommon **T**ype **S**ystem (**CTS**) and the **C**ommon **L**anguage **S**pecification (**CLS**).

- *Language independence.*

Since we heavily use this feature in our architecture, we will try to investigate it deeper than the others. As we already mentioned, the .NET Framework introduces the CTS and the CLS. Common Type System is a specification for how types are defined, used and managed in the runtime. It establishes a framework that helps to provide cross-language integration. It also defines the rules that languages must follow, so that objects written in different languages can interact with each other.
The Common Type System supports two general categories of types: value types and reference types. Value types contain the data or value whereas reference types store a reference to the memory address of the value.
Common Language Specification is a set of basic language features that are needed by many applications. The rules that apply to the Common Type System, apply to CLS as well, but the rules of CLS are more strict. CLS ensures interoperability by defining a set of features that developers can depend on and which are available in a wide variety of languages.
Thanks to Common Type System and Common Language Specification, the .NET framework supports language interoperability, cross-language integration, language independence, etc.
To complete the overview of the types, it should be noted that CLR provides a set of primitive types that all languages support. The data types include: `Integer` (three types: 16/32/64 bits), `Float` (two types: 32/64 bits), `Boolean`, `Character`, `DateTime`, and `Time Span`.

- *Base Class Library.*
  The **B**ase **C**lass **L**ibrary (**BCL**) is the core of the Framework Class Library. It is a library of functionalities available to all the languages which use the .NET Framework. The BCL provides classes that encapsulate a number of common functions, including file reading and writing, graphic rendering, database interaction, XML document manipulation, and so on. It consists of classes and interfaces of reusable types that integrate with Common Language Runtime.

- *Security.*
  The design addresses some of the vulnerabilities, such as buffer overflows, which have been exploited by malicious software. Additionally, .NET provides a common security model for all applications.

- *Portability.*
  The framework has been engineered to be platform-agnostic, and cross-platform implementations are available for other operating systems. Microsoft submitted the specifications for the Common Language Infrastructure, the `C#` language, and the `C++/CLI` language making them available as official standards. This makes it possible for third parties to create compatible implementations of the framework and its languages on other platforms.

The execution process of .NET framework is the following. First of all, the code written in any language that is supported by the framework is compiled by its appropriate compiler. The compiled code is known as Intermediate Language code, which is also known as managed code.
The managed code is then passed through the CLR where it is again compiled and converted to object code (or native code) by the Just In Time compiler. The native code is then passed to the hardware for the execution.
The runtime manages the references to objects and automatically handles the object layout. It performs memory management by releasing the objects that are no longer needed. This handling of objects is known as managed data.
Garbage collection eliminates the memory leaks as well as some other common programming errors. If the code is managed, we can use managed data, unmanaged data, or both in our .NET Framework application.

Inside the .NET framework, the **W**indows **P**resentation **F**oundation (**WPF**) is the main technology used to develop the software user interface. WPF introduces a new XML-based language to represent the user interface, known as **XAML** (e**X**tensible **A**pplication **M**arkup **L**anguage). In XAML, elements are represented as XML tags. XAML allows applications to dynamically

parse and manipulate user interface elements both at compile-time and at runtime, providing a flexible model for user interface composition.

2.5.1 Custom Attributes and Reflection

This section deserves particular attention because it describes two fundamental concepts we heavily relies on in this work: the *Custom Attributes* and the *Reflection* concepts, from the .NET point of view.
As we did for the previous section, we explain the concept here in order to be able to just recall it later on in the development chapter, avoiding then the opening of a parentheses which would be long and hard to understand.

The starting point to explain these two concepts is the *Metadata* notion. A .NET application can be viewed as an application containing code, data, and metadata. Metadata is information about the code and the data (like information about the types, code, assembly, and so on) that is stored along with your program. Metadata, in the Common Language Infrastructure (CLI) we have already mentioned, refers to certain data structures embedded within the Common Intermediate Language code (CIL) that describes the high-level structure of the code. It describes all the classes and class members that are defined in the assembly. The metadata for a method, for instance, contains a complete description of the method, including the class (and the assembly that contains the class), the return type and all of the method parameters.
A CLI language compiler generates the metadata and stores this in the assembly containing the CIL. [Figure 2.11]
Conceptually metadata is characterized by the fact that .NET has a kind of database which stores all the types and methods, just above the bytecode. When we compile a file, then, we obtain an assembly that is structured as the previous mentioned Figure 2.11.

Most metadata is generated automatically by the compiler, but custom metadata can be added to our programs using Attributes. The process by which

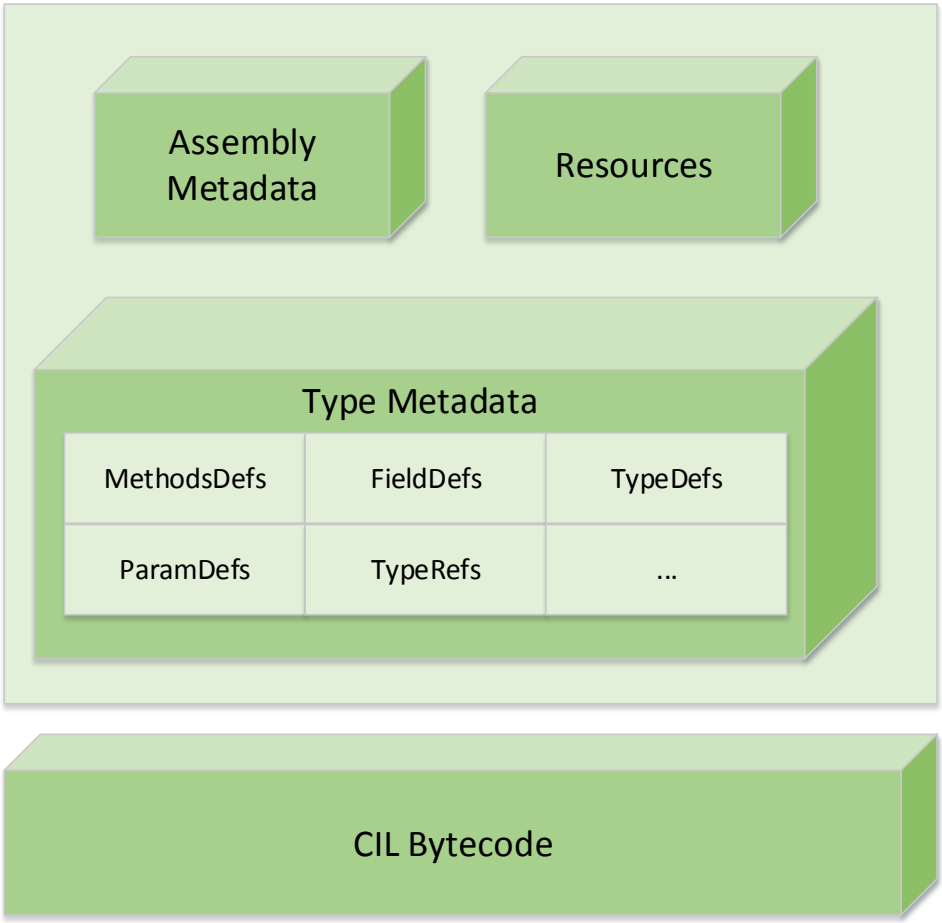

Figure 2.11: The .NET Assembly Structure

a program can read its own metadata is supplied by what is usually called Reflection. Let's start from this last concept first.

Reflection is the ability of a computer program to access metadata at run-time, making it available somehow, without knowing it at compile time. The classes in the Reflection namespace, along with the `System.Type` and `System.TypedReference` classes, provide support for examining and interacting with the metadata. Reflection can also be used to adapt a given program to different situations dynamically.

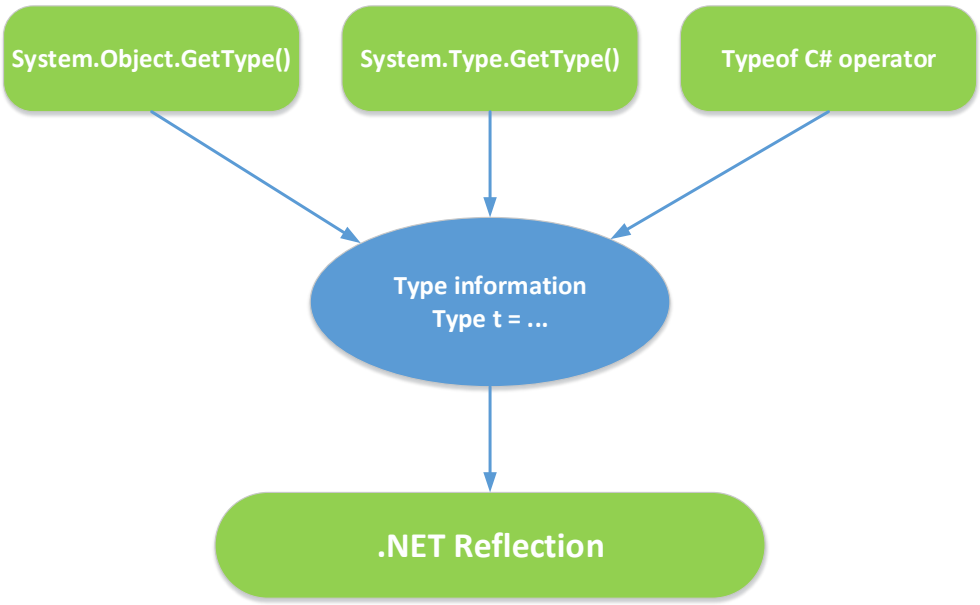

Figure 2.12: .NET Reflection

Through Reflection a program can reflect on itself, extracting metadata from its assembly and using it in different ways, e.g. discovering an assembly on the user's machine, querying what methods are available and invoking one of those members dynamically. Using Reflection, you can get the same information you can see in a class viewer.
.NET Reflection behaves basically like the Java one, and it can be used in three different way to obtain a Type reference: [Figure 2.12]

- *Using* `System.Object.GetType()`:
  It returns a Type object that represents the type of an instance.

- *Using* `System.Type.GetType()`:
  Another way of getting type information which gets the type with the specified name, performing a case-sensitive search.

- *Using the* `typeof()` *Operator in* `C#`:
  The last way to obtain type information. This operator takes the name of the type as a parameter.

The Type class, as you can see, is the root of the reflection classes. Type encapsulates a representation of the type of an object. The Type class is the primary way to access metadata.

Reflection is generally used for different tasks: [Lib01]

- *Viewing metadata*: This might be used by tools and utilities that wish to display metadata.
- *Performing type discovery*: This allows you to examine the types in an assembly and interact with those types or instantiate them.
- *Late binding to methods and properties*: It allows the programmer to invoke properties and methods on objects (in particular on those which were dynamically instantiated using type discovery).
- *Creating types at runtime*: It is useful to create new types at runtime and then use them to perform tasks.

.NET reflection has something which makes it even more powerful: it can be extended using special annotations called *CustomAttributes*.

To see what they stand for, let's start from the simple Attribute concept in .NET.
An attribute is an object that represents data we want to associate with an element in our program. The element to which we attach an attribute is referred to as the target of that attribute.
Attributes come in two flavors: *intrinsic* and *custom*. Intrinsic attributes, as the name itself suggests, are part of the CLR and are integrated into .NET. The prime example of intrinsic attribute is given by the `[WebMethod]` intrinsic attribute. By using just this attribute, we indicate that we want the method exposed as part of the XML Web service, saving a lot of code and time. Another noteworthy example of intrinsic attribute is the popular `[Serializable]` attribute, which allows a class to be serialized.

Custom Attributes are attributes which we create for our own purposes. They don't alter the semantics of the language and can be a powerful tool when combined with reflection. It is possible to define several custom attributes in our projects. They will be added together with the specified attributes in metadata. We need to tell the compiler with which kinds of elements this attribute can be used: they represent what we call the attribute targets.
The possible attribute targets are: `All, Assembly, Class, ClassMembers, Constructor, Delegate, Enum, Event, Field, Interface, Method, Module, Parameter, Property, ReturnValue, Struct`.
By defining a target, we prevent the system to use an attribute in an invalid way: e.g. if the target is class the system prevents us from using it for methods, and so on.
It is possible to apply attributes to their targets by placing them in square brackets immediately before the target item (e.g. the class declaration). Creating a custom attribute is possible by deriving our new custom attribute class from System.Attribute. In this way we are also instructing the Runtime to store this attribute in the metadata.

To see how custom attributes work, it is better to give an example. [Listing 2.3]
Let's suppose that we want to create a `DescriptionAttribute` custom Attribute and then use it by assigning it to the `MyClass` class. The process is very simple and self-explanatory and can be done as follows:

Listing 2.3: Custom Attribute Example

```
1 namespace Customs_Test
2 {
3    using System;
4    using System.Reflection;
5
6    // create custom attribute to be assigned to class members
7    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method|
8        AttributeTargets.Property, AllowMultiple = true)]
9    public class DescriptionAttribute : System.Attribute
```

```
10    {
11       private int descrID;
12       private string descr;
13       private string submitter;
14
15       // attribute constructor for positional parameters
16       public DescriptionAttribute(int descrID, string submitter)
17       {
18          this.descrID = descrID;
19          this.submitter = submitter;
20       }
21       // property for named parameter
22       public string Descr
23       {
24          get { return descr; }
25          set { descr = value; }
26       }
27    }
28
29    // ********* assign the attributes to the class ********
30    [DescriptionAttribute(12,"Nadia Vetrano")]
31    [DescriptionAttribute(15,"Nadia Vetrano",
32                             Descr="Test Description")]
33    public class MyClass
34    {
35      //......
36    }
37 }
```

The convention is to append the word Attribute to our attribute name (line 9). The compiler, however, also allows us to call the attribute with the shorter version of the name.

Attributes can take two types of parameters: positional and named parameters. In the Description example, the descrID and the submitter are positional parameters, while descr is a named parameter. Positional parameters are passed in through the constructor and must be passed in the order declared in the constructor. Named parameters are implemented as properties. Once we have defined the attribute, we can use it by placing it immediately before its target (lines 30-31). In this way, it will be stored with the metadata.

At this point, we can use the C# Reflection support to read the metadata in the MyClass class. [Listing 2.4]

Listing 2.4: Custom Attribute Retrieving Properties Example

```
1  public static void Main( )
2  {
3      MyClass mm = new MyClass( );
4      System.Reflection.MemberInfo info = typeof(MyClass);
5      object[] attributes;
6      attributes =
7         info.GetCustomAttributes(
8             typeof(DescriptionAttribute), false);
9      foreach(Object attribute in attributes)
10     {
11         DescriptionAttribute da = (DescriptionAttribute) attribute;
12         Console.WriteLine("\nDescrID: {0}", da.descrID);
13         Console.WriteLine("Submitter: {0}", da.submitter);
14         Console.WriteLine("Descr: {0}", da.Descr);
15     }
16 }
```

The output of [Listing 2.4] is:

```
DescrID: 12
Submitter:  Nadia Vetrano
Descr:

DescrID: 15
Submitter:  Nadia Vetrano
Descr:  Test Description
```

We start by initializing an object of the type MemberInfo (System.Reflection, line 4). This object is provided to discover the attributes of a member and to access the metadata. It derives from Type and encapsulates information about the members of a class (e.g., methods, properties, fields, events, etc.). This means, we get the member information and use it to retrieve the custom attributes. We then call the typeof operator on the MyClass type, which returns an object of type Type, which derives from MemberInfo (line

7).
The next step is to call `GetCustomAttributes` on this object, passing in the type of the attribute we want to find. What we get back is an array of objects (line 6), each of type `DescriptionAttribute`.
We can now iterate over this array, retrieving and printing out the properties of the `DescriptionAttribute` object.

2.6 AvalonEdit

AvalonEdit is the name of the new WPF-based text editor written for SharpDevelop 4.0. [Prob]
We use it as the starting point of our Interactive Clips window. The tool allows the project to have a text editor that is extensible and easy to use.
The main class of the editor is the `AvalonEdit.TextEditor` class and it is possible to use it just like a normal WPF `TextBox`:

```
<avalonEdit:TextEditor
    xmlns:avalonEdit=
        "http://icsharpcode.net/sharpdevelop/avalonedit"
    Name="textEditor"
    FontFamily="Consolas"
    SyntaxHighlighting="C#"
    FontSize="10pt"/>
```

AvalonEdit has syntax highlighting definitions built in for several languages like: `ASP.NET, C++, C#, HTML, Java, JavaScript, PHP, VB`, and `XML`. The example above shows the definition of an AvalonEdit Text Editor for a `C#` application. Since we need the Clips syntax highlighting definitions for our project, we did not restrict ourselves to merely using the tool but we extended it instead.
We will discuss this in depth later on in Chapter 3, but here we want to describe the tool in order to give the reader the means to understand it. The AvalonEdit architecture is summarized in Figure 2.13

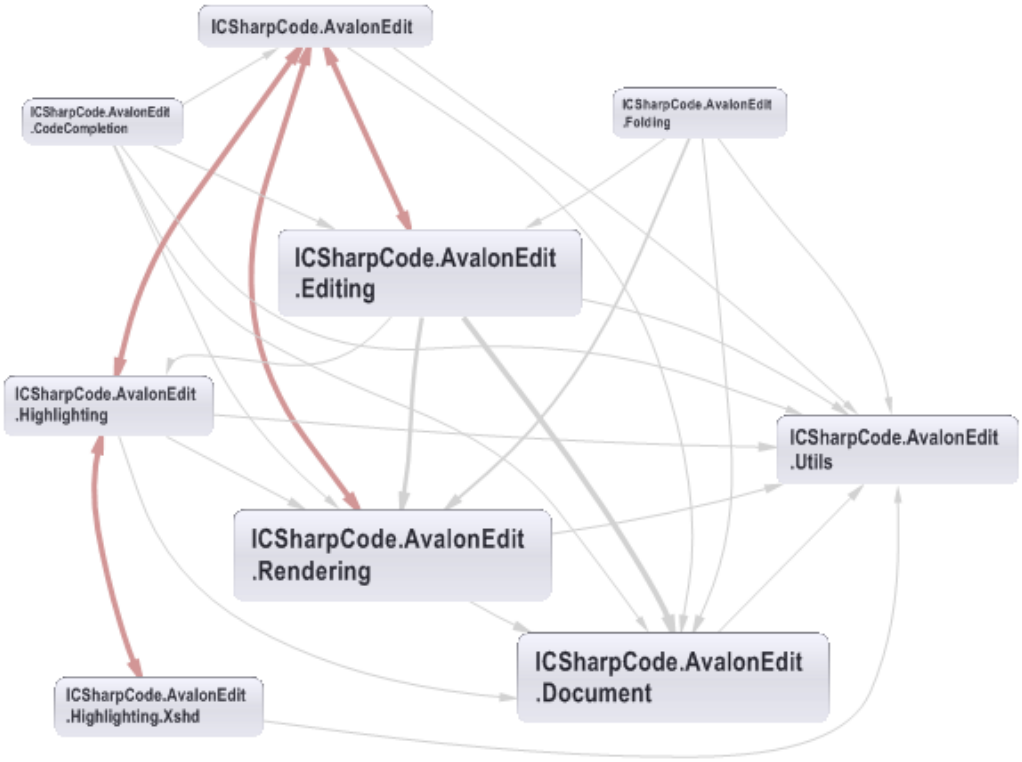

Figure 2.13: The AvalonEdit Architecture

The AvalonEdit tool is a composite control with three layers:

- `TextEditor`: It's the main control.
- `TextArea`: used for editing.
- `TextView`: used for rendering.

While the main control provides some convenience methods for common tasks, for most advanced features, you have to work directly with the inner controls: you can access them using `textEditor.TextArea` or `textEditor.TextArea.TextView`.
The main class of the model is `AvalonEdit.Document.TextDocument`. Basically, the document is a `StringBuilder` with events. However, its namespace also contains several features that are useful to applications working with the text editor.

In AvalonEdit, an index into the document is called an *offset*, and is exactly the same as the *index* parameter used by methods in the .NET `String` or `StringBuilder` classes. Offsets are easy to use, but it is also possible to define a struct called `TextLocation` for the same purpose.

The two main features of AvalonEdit we used in this work are the Syntax Highlighting and the Code Completion.
The highlighting engine in AvalonEdit is implemented in the `DocumentHighlighter` class. Highlighting is the process of taking a `DocumentLine` and constructing a `HighlightedLine` instance for it, assigning colors to different sections of the line. The `HighlightingColorizer` class is the only link between highlighting and rendering. It uses a `DocumentHighlighter` to implement a line transformer that applies the highlighting to the visual lines in the rendering process.

The rules for highlighting are defined using an "e **X**tensible **S**yntax **H**ighlighting **D**efinition" (.xshd) file.
We can see a simplified example of the one used for `C#` in Figure 2.14. This file looks similar to an XML file, and consists of the definitions of colors for keywords, comments, Strings, etc., followed by a set of rules and regular expressions which tell you how and where to apply these colors. As we said before, several languages already have built-in syntax highlighting definitions, but our language, Clips, is not in this list (quite obviously). This means that we had to create a new `.xshd` file from scratch, along the lines of the existing ones. We will see this file in Chapter 4.

Let's now see the main points of a general `xshd` file.

The highlighting engine works with *spans* and *rules* that specify a color assignment.
Spans consist of two Regular Expressions (begin+end), while rules are simply a single regex with a color. The <`Keywords`> element is just a nice syntax

```
<SyntaxDefinition name="C#"
        xmlns="http://icsharpcode.net/sharpdevelop/syntaxdefinition/2008">
    <Color name="Comment" foreground="Green" />
    <Color name="String" foreground="Blue" />

    <!-- This is the main ruleset. -->
    <RuleSet>
        <Span color="Comment" begin="//" />
        <Span color="Comment" multiline="true"
           begin="/\*" end="\*/" />

        <Span color="String">
            <Begin>"</Begin>
            <End>"</End>
            <RuleSet>
                <!-- nested span for escape sequences -->
                <Span begin="\\" end="." />
            </RuleSet>
        </Span>

        <Keywords fontWeight="bold" foreground="Blue">
            <Word>if</Word>
            <Word>else</Word>
            <!-- ... -->
        </Keywords>

        <!-- Digits -->
        <Rule foreground="DarkBlue">
            \b0[xX][0-9a-fA-F]+  # hex number
        |    \b
            (    \d+(\.[0-9]+)?   #number with optional floating point
            |    \.[0-9]+         #or just starting with floating point
            )
            ([eE][+-]?[0-9]+)? # optional exponent
        </Rule>
    </RuleSet>
</SyntaxDefinition>
```

Figure 2.14: A simplified xshd file for `C#`

to define a highlighting rule that matches a set of words; internally, a single regex will be used for the whole keyword list.

The highlighting engine works by first analyzing the spans: whenever a begin regex matches some text, that span is pushed onto a stack. Whenever the end regex of the current span matches some text, the span is popped from the stack. Each span has a nested rule set associated with it, which is empty by default. This is why keywords won't be highlighted inside comments: the span's empty rule set is active there, so the keyword rule is not applied. This feature is also used in the string span.

What's great about the highlighting engine is that it highlights only on-

demand, works incrementally, and yet usually requires only a few KB of memory even for large code files: the memory usage of the highlighting engine is linear to the number of span stack changes, not to the total number of lines. This allows the highlighting engine to store the span stacks for big code files using only a tiny amount of memory.

Let's now describe the Code Completion feature.
AvalonEdit comes with a code completion drop down window. The programmer needs to handle the text entering events to determine when the window should be shown, but all the user interface is already done for him/her. As for the previous feature, we will see how we used it by going over the code in Chapter 4.
In the editor, the control will open the code completion window whenever `'.'` is pressed. By default, the `CompletionWindow` only handles key presses like Tab and Enter to insert the currently selected item, but it is also possible to make it complete when keys like `';'` are pressed.
The `CompletionWindow` will actually never have focus. This means it hijacks the WPF keyboard input events on the text area and passes them through its `ListBox`. This allows selecting entries in the completion list using the keyboard and normal typing in the editor at the same time.

Chapter 3

ARCHITECTURE DESIGN

Starting from the FACE project, the main goal of the iClips system is to extend the cognitive part of the robot through the introduction of a modular architecture and an expert system which allows to coordinate, control and debug the android interactively.

3.1 An Hybrid Architecture

The architecture we propose is a hybrid robotic control architecture organized into layers and we consider it as the cognitive part of the robot.
The main reason behind this decision lies in the fact that humans can be considered sophisticated autonomous agents that can operate in complex environments through a combination of reactive behaviour and deliberative reasoning. Since we are working in the humanoid robotics field, this observation led us to propose an *hybrid deliberative-reactive* architecture, which combines a behaviour-based reactive layer and a logic-based deliberative one. A behaviour-based reactive layer ensures that the robot can handle the various real-time challenges of its environment appropriately, while a logic-based deliberative module provides the agent with the ability to perform more complex high-level tasks that require planning. [QTG04]
The proposed architecture addresses the critical challenge of combining reactivity and deliberation.

3.1.1 The Deliberative/Reactive Paradigm

A robotic paradigm can be described by the relationship between the three commonly accepted primitives of robotics: SENSE, PLAN, and ACT. [Mur00]

If a function acquires information from the sensors of the robot and produces output which is useful for other functions, then it falls in the SENSE category. If it acquires information (either from sensors or its own knowledge about the world) and produces one or more tasks for the robot to perform (turn left, smile then blink), that function is in the PLAN category. Instead, functions which give output commands to actuators fall into the ACT category (turn 20°) clockwise with a turning velocity of 0.1mps).

The Hybrid Deliberative/Reactive Paradigm on which we focus is one of the three common paradigms (Hierarchical, Reactive, Hybrid) used in robotics, and it is also the latest one in terms of time.
The Hierarchical Paradigm comes first and relies much on planning. Under it, the robot senses the world, plans the next action, and then acts (SENSE, PLAN, ACT in sequence). Then it repeats this sequence of activities again and again, as long as it likes.
The Reactive Paradigm was a reaction to the Hierarchical one. It threw out planning by using just a SENSE-ACT type of organization. While the Hierarchical Paradigm assumes that the input to an ACT paradigm is the result of a PLAN, the Reactive one assumes that the input to an ACT is the direct output of a SENSE paradigm. But it quickly became clear that throwing away planning was not a good idea for general purpose robots: this is why the Reactive Paradigm served as the basis for the Hybrid Deliberative/Reactive Paradigm we chose to use.
Furthermore, behaviours in the Hybrid paradigm have a slightly different meaning than they have in the Reactive one. In the Reactive paradigm, a *"behaviour"* indicates a purely reflexive behaviour, while in the Hybrid paradigm, the term is nearer to the *"skill"* concept (that means more like a "learnt behaviour").

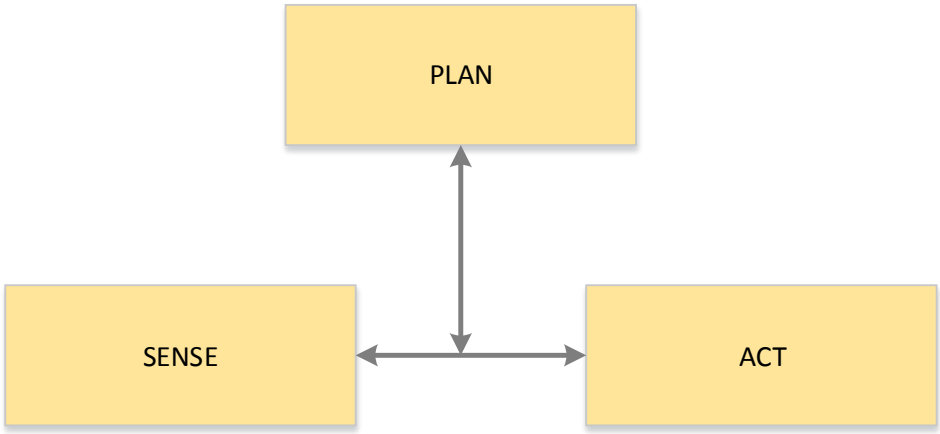

Figure 3.1: The Hybrid Deliberative/Reactive Paradigm

Under the Hybrid Paradigm, the robot first plans (deliberates) how to best decompose a task into subtasks and then what are the suitable behaviours to accomplish each subtask. Then the behaviours start executing as per the Reactive Paradigm. This type of organization in the Hybrid Paradigm is PLAN, SENSE-ACT. Planning is done at one step, sensing and acting are done together [Figure 3.1]. The system is conceptually divided into a reactive layer and a deliberative layer.

The guideline is that functions which operate on symbolic information go to the deliberative layer, while functions which transform sensor data into actuator commands go in the reactive one.
A Deliberative/Reactive Paradigm is often the best architectural solution for several reasons. First it allows deliberative functions to execute independently of reactive behaviours (e.g. a planner can be slowly computing the next goal for the robot while it is reactively navigating towards its current goal). Second, such an architecture usually encapsulates functionality into modules and this allows subsystems or software agents to be mixed for specific applications. [AM94]

Our two-layer Deliberative/Reactive architecture is highly modular. Modules however have essentially a functional structure: they collect information and act based on them, then they bring them back up to the deliberative part.
In such a hybrid architecture, the Clips expert system finds a natural application.
Modules can assert high-level properties and rules can call actions exposed by these modules. This implicitly creates a deliberative-reactive architecture where the reactive part is carried out by modules, while the deliberative part is designed above them.
Then we have the libraries level -above- (the libraries used from the application, e.g. Kinect SDK), and a bunch of modules -below- using these libraries. These modules can put the input into effect, read it and potentially make a first step of abstraction before passing data to the deliberative level (e.g. give the same identifier to the same person entering the scene if he/she has already been seen before).
It should be clear that with the introduction of Clips, we divide the world in two part: one is a world of symbolic reasoning which gives us the control on the deliberative aspects, the other one is a world organized in modules which gives us the control on the reactive aspects, and we can decide where to predicate. In the symbolic reasoning world everything should be *reified* in terms of functions and assertions: at that point it's possible to decide which is the symbolic level we want.
In such a world, an assertion could correspond to an information like *"I've read the raw sensor, there is a person"* or it could corresponds to an information like *"there's Leonardo, I've already seen him"*.
This means that regarding the assertion level the architecture lets the user to decide where to put the imaginary *boundary* which divides the two layers.

3.2 System Requirements

The developed architecture realizes essentially a software refactoring of the FACE system and it is designed to meet several requirements. In order to accomplish them, we focused on several different aspects.

One of the fundamental requirements of the system is to be *debuggable*. It must be possible to *"observe"* the contents of the robot brain and its execution through an interactive tool which allows to suspend the robot, observe it, force it to do something, interactively try behaviours, and so on.
The Clips expert system turned out to be fully suitable for this purpose. It allows the user to debug the application interactively, interrupt its execution and examine its internal state. This makes the FACE brain content observable and manipulable at run-time.

Another important aspect is that the iClips system is designed to be *standalone*. That means we actually realized a debugger for modules without being forced to always use the robot for real.

Going on with the requirements, we meet the one which is most important: the system must be *declarative*.
This is required so that the system can be used by *hybrid subjects*, that are people who probably don't have programming skills. As we already know, the tool we developed allows the system to perform symbolic abstraction: this declarative (as opposed to procedural) way to express knowledge makes it possible for hybrid users to encode rules for attitudes and behaviours without computer programming experience. One of the main challenges we want to tackle is the development of a framework which supplies people who have to write modules for the FACE robot and debug them with an architecture and a guided environment. The key point is that *working on the module level requires computer technicians and scientists, while working at the Clips level only requires people trained to write simple rules.*
Another advantage of the rules is that rules are additive. Rules are indepen-

dent from each other. People who want to add a new rule do not need to be aware of how the rest of the system is been designed and implemented. The significant outcome is that almost everyone can work on the system by adding information.
Rules, in fact, follow the very intuitive `if-then-else` pattern.
It could be asserted that as long as people only have to produce such structured rules, no problem arises: this paradigm proves to be something that almost everybody can use. (e.g. this is confirmed by the spread of websites gaining market like the *IF-THIS-THAN-THAT* website [IFT]).

The main causes of troubles and distress for people are constructs like the iteration, the "foreach" and other constructs that are familiar to computer scientists but not to the average user. If we build adequately abstract levels (as a rule system), we are offering an abstraction that is certainly more user-friendly for those unfamiliar with this field. From the point of view of FACE, since the development of such a system is half-way between the technical and the biomedical part, this is a good starting point as can establish that *"Up to here there are the computer scientists, then there are psychologists or someone else on their behalf"*.

A further requirement of the system is that it can be seen as *a blackboard*, a tool in which everyone can write and read. This is how the Clips model itself is designed and it implies that modules are independent to each other (there are no functional dependencies) and act "directly" through Clips rules. In this way, if a module isn't loaded (and so it contributes no facts), rules can predicate on it without damages or problems: a model like this realizes the desired coordination.

Going further with requirements, a noteworthy aspect is represented by the iClips interface. Although it is the outcome of a chain of challenges we faced (we will argue this point later on), for the user it represents a way to observe the robot behaviour through a graphic environment. So after the design of the architecture, we worked on the interface (a specific section will provide the details regarding the interface).

The last requirements is that the system is *adaptive* and *interactively extensible*.

We started from a monolithic architecture, but we wanted to make it modular. That means to build an "open" architecture, in order to make it possible to add, remove, modify, and select modules as needed. So what we needed was the opposite of a preconfigured architecture, according to all the other requirements.
The robot does not have just a single operation, hence it should be adaptive and always "ongoing". It cannot be a machine in which we compile first and then we see the outcome in terms of behaviour and attitudes. Instead it needs to be a deliberative system, capable of having a clear representation of its targets (e.g. teaching emotions to children).

Another important aspect to point out is that in our architecture we deal with complex data structures. In such a context, the CLIPSNet Library (section 2.1.4), that is not an automatic wrapper, does the job properly. The library is an handmade wrapper, and it comes out to be very useful in handling such structures. The outcome is that the quality of the interface to Clips is high, compared to what it could have been if the code had been written by an automatic wrapper generator.
Due to all these reasons, the iClips system can use the CLIPSNet Library as a good facility. This allows the user to interact with Clips without having to know how the particular wrapper we are using is made.

Thanks to the CLIPSNet Library the whole architecture perceives the *InteractiveClips* (iClips) as Clips itself.

3.3 System Overview

In this section we aim to provide an overview on the architecture we developed.
We can see it as composed by three blocks: the libraries, the modules, and

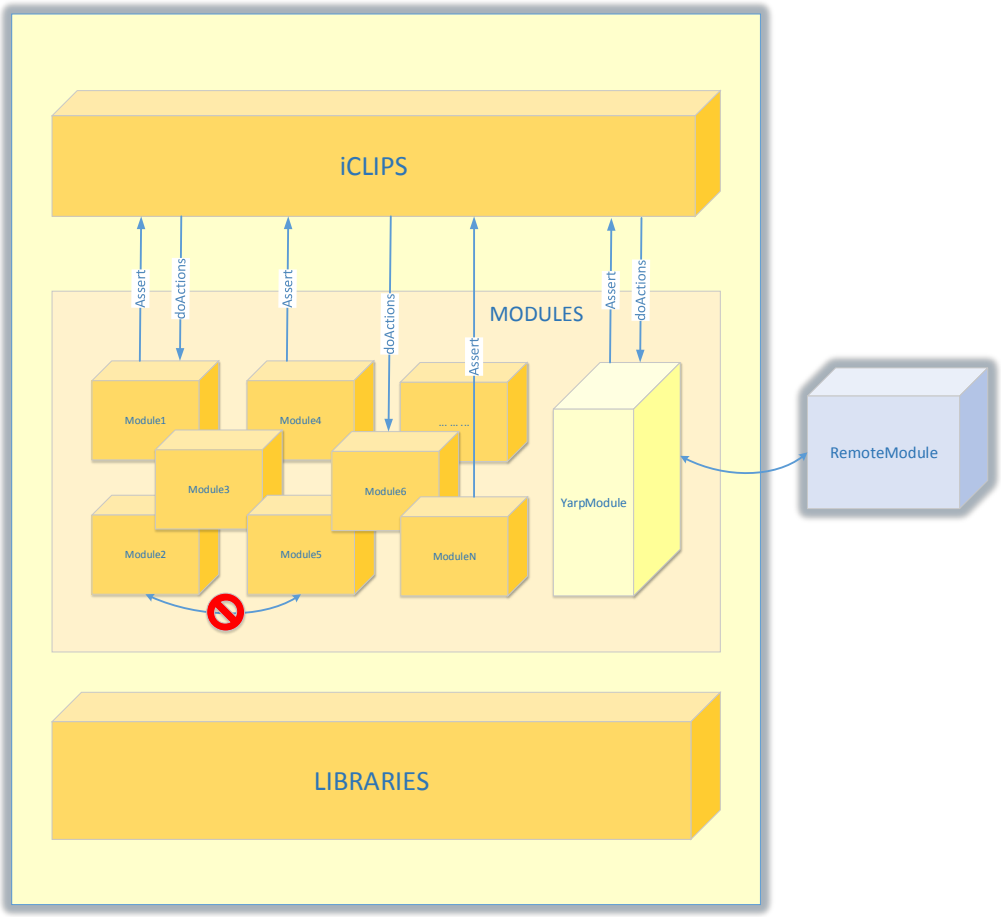

Figure 3.2: The Architecture

iClips. [Figure 3.2]

The Libraries Section can be viewed as the lower level of the architecture[1]and is made up by all the libraries of the tools we want to use on our system. The section might include for instance the SDK Kinect Library, the FACE Libraries or any other library as well.

Between this block and the iClips one, there is a potentially large section including all the modules of the architecture. In fact, an essential contribution of this work consists of making the existing monolithic architecture a

[1]when saying "lower" we just mean conceptually in the design of the architecture. The term is not intended as related with its weight on the architecture.

modular one, so that modules are kept independent.
This aspect is a major priority and represents part of the reason we designed such architecture. Having single distinct modules guarantees that the system can accomplish several goals such as code maintainability or functional independence: e.g. if at some point we decide to replace the Kinect sensor (section 2.3) with a brand new one, we should be able to do so just by replacing the relative module, that means with the least possible effort.
For the same reason modules don't speak directly to each other. Their communication needs to be managed by the expert system Clips, which represents the upper level of the architecture.
The iClips block, which uses the CLIPSNet wrapper (section 2.1.4) to realize -among the other things- the iClips editor for debugging and interaction, is the heart of the modular architecture, because it is its coordinator. Each module we mentioned communicates to Clips through assertions, and the expert system owns the appropriate rules to trigger the actions to be executed, if any.
Looking at the architecture, it should be emphasized that it can contain as many modules as we want, conversely, there is only one Clips environment. Modules are orchestrated by the `ModuleManager`, wich is an important element located in the iClips block.
In the final analysis, the iClips block is a module itself, and part of the architecture. However, since we imagine each module in the intermediate level communicating with Clips bidirectionally, we want to emphasize its importance by placing it as the upper section of the design structure. For the same reason, we refer to the architecture as the *"iClips System"* : this name underlines the main contribution iClips gives to the architecture, both with respect to the expert system and to the modular approach it gave rise to.

Through iClips, we get Clips plus the possibility of having an editor for interaction. At this point it should be clear that the iClips name stands for *Interactive Clips*. Let's spend a few words about its origin.
The idea comes from the introduction in the architecture of the expert system Clips, initially integrated and used directly from `C#` through the CLIPSNet

wrapper. This first step made it possible to use the expert system for general purposes in our work, but it did not provide all the functionalities which now constitute the strong points of the system. That is, in the architecture there was no way to interact with the robot at runtime, or to make it observable and debuggable. Also, there was no way to add new modules, or to keep them independent, or to expose actions to Clips easily, and so on.
This is why the concepts of graphic interface and *"What you see what you get"* editor came in our mind. We decided to develop a window to be used as a view of the rules world. This allows the hybrid user to interact with the system and use it without any programming skills. All the average user needs is a basic knowledge of this rules world (section 3.2).

This is how iClips was born.
The next step was to provide the system with a dialog window with the same features of a syntax-driven editor, like Syntax Highlighting and Code Completion. This deserves a specific section in paragraph 3.4

The main goal has always been to have a *Declarative Interface*.
By reaching this goal, we get the optimal setting which allows both the user and the programmer to work easily: the user does not need to think too much when using the system; the programmer does not need to be aware of every detail when adding a new module. In fact, developers who write new modules need to know as little as possible of the architecture: they just have to be aware of some simple procedures we developed to easily let them expose methods to Clips, assert facts on it and load the new module.
These functionalities represent a big advantage when writing/adding new modules, and they are achieved by using the .NET Custom Attributes we already discussed in Section 2.5.1.
We can now see the Custom Attributes from the point of view of our system. If we did not have Reflection and Custom Attributes in the architecture, it would not have been possible to have these new functionalities as simple as we described. By using Reflection in fact, we have been able to have our own notations and, due to this, to build our little *"Domain Specific Language"*.

The two main Custom Attributes we introduced are the `ModuleDefinitionAttribute` and the `ClipsActionAttribute`.
The `ModuleDefinitionAttribute` allows the dynamic loading of modules in the system, the `ClipsActionAttribute` creates a way of exposing methods to Clips by writing a simple annotation just before their definition. This is a very intuitive and powerful tool, and it takes care of both exposing the method to Clips and adding the method's signature to the iClips editor CodeCompletion.
Thanks to this attribute, you do not need to write specific handmade CLIPSNet delegates any more, as we described in paragraph 2.1.4.

The declarative interface we managed to realize keeps the pace with the Custom Attributes we introduced: by using them, all the modules need to know is that they have to assert facts, expose actions and that's it. Once you've done so, you can assert facts from `C#` in the rules world and from this last one you can call opportunely annotated `C#` methods.
Because of this, the Interactive Clips is then an integral and essential part of the architecture. It completes these new tasks while behaving like the expert system Clips itself, and represents the coordinator of the whole architecture.

3.3.1 Modules Template and Behaviour

All of the modules are structured in the same way, no matter what they contain or what are they used for.
We will examine this aspect more deeply by giving examples in the next chapter, when we will focus on the Modules Implementation. A concrete example with code will make it easy to explain how we decided to design the Modules template. In this section we will point out some aspect of their structure, which should allow them to be understood better.

There are two main concepts: the `ModuleManager`, and the Modules themselves.
The Modules Orchestrator, as we said, is represented by the `ModuleManager`

in the iClips module. It owns the Clips environment and takes care of orchestrating the modules which share this environment.
The idea is that the `ModuleManager` combines the modules, and the Clips environment is the only object that is shared between modules. The iClips infrastructure is built by giving to the `ModuleManager` the possibility of loading modules and unloading them (if necessary) dynamically. The class which contains the `ModuleManager` is also in charge of creating the Clips environment, passing it to the `ModuleManager` and running the rule engine.

Module independence is of primary importance in such a structure. The dynamic loading of modules is strictly related to this concept.
As it is outlined in Figure 3.2, modules express facts by asserting them on Clips. Each module does not see the other modules, it merely asserts its own facts and receives commands from the rules that have been triggered, if any. Modules are written in such a way to assert things and receive actions: that's it.
The only way in which two modules can interact is the existence, in the rule engine, of rules that involve more that one module. In any case, modules don't interact directly.
E.g. ModuleAudio and ModuleVideo assert facts, a compound rule use some of these facts in its antecedent and maybe makes a third module, ModuleRobot, do something.
In this way we can write independent modules and prevent the introduction of dependencies which would cause the system to regress towards a monolithic solution.

A module, however, may need other modules to be loaded.
We implemented this feature by introducing the "`require`" keywords. We defined a Clips function `require` that basically says: *"if the module moduleName isn't loaded, then load it, else do nothing"*. In this way, if the moduleA requires moduleB to work, we can just write inside the moduleA clp file: "`require ModuleB`". If moduleB is already loaded in the system, it is not reloaded again, otherwise the `loadModule` method is invoked.

We can use the `require` keyword both from the clp file of the modules and from the iClips Editor.

The first step to build a new module is to derive it from the Module class: then we will have it available and ready to talk with Clips without having a clue of the particular wrapper we are using.
Now, what about the structure of a module? The idea is the following. We have several modules and they are compiled separately. Each module has a script which allows it to load its own clp file.
This means that when a module is loaded by the `loadModule(testModule)` function, its `init.clp` is loaded and the `C#` methods which have been annotated with `[ClipsAction("ActionName")]` are loaded too.
We should remind that it's mandatory for these methods to use the `Clips-net.Datatype`.

We keep our modules in a Module directory inside the executable. The Module directory contains a folder for each module, and each folder contains the appropriate dlls.

For example if we have a testModule, its relative path inside the executable is `Modules\testModule\`. Then, the `testModule` directory has potentially several dll inside. Using reflection, we look for the dll which contains the definition of the subclass of Module. At this point the `testModule` has been loaded, and we can use the exposed clipsActions. The related code is then executed in `C#`.
So, when we ask the `ModuleManager` to perform a `loadModule`, it first looks for the right dll through the `FindModule` method via Reflection, in order to ensure it exists. The directory might contain several dll, so this is the moment to look for the right one.

Summarizing, each module directory contains at least two files: a `dll` and a `.clp`. The clp file is used to list which other modules are needed for this module to work properly. The iClips will ensure that they are loaded. That means if you wish to load a module, all you need is to edit the specific `.init`

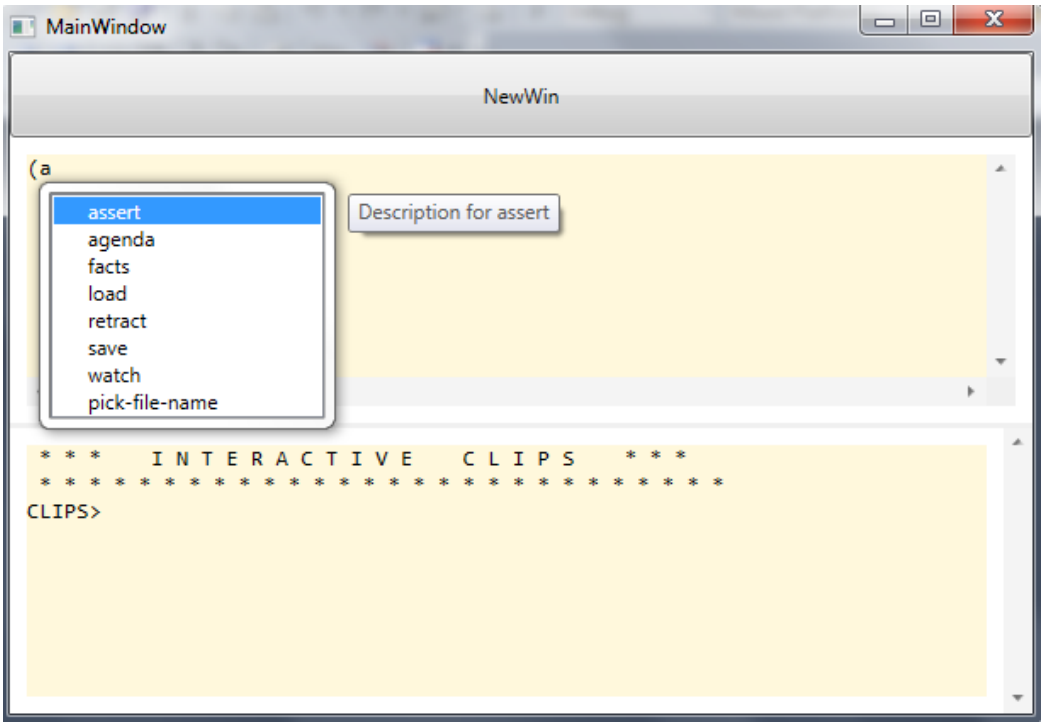

Figure 3.3: The iClips Editor

file.

If the user does not provide a new name for the clp file (through the `ModuleDefinitionAttribute`), the system names the .clp after the module. If he forgets to create it at all, the system creates it for him and names it as we described. By doing this, the system attempts to avoid problems and crashes.

3.4 iClips Interface

Since we are describing the design of the architecture, a special mention goes to the graphic interface which allows the user to interact with the system from the Clips point of view. This declarative interface is represented by the iClips syntax-driven editor, and it is one of the key points of this work. [Figure 3.3]

We can imagine the iClips window as a *"What you see what you get"* edi-

tor, which allows the user to have a window from FACE to the rules world. By building such an interface, we want to create something that enables the users to have the control during the testing phase.
This window/console can be present during the execution of the program. This allows the user to interface with the Clips world in a simple and intuitive way. Thanks to it, the user can assert new facts, check the existing ones, add new rules, evaluate the existing ones (and so on), by using the same commands as he would have used in the Clips world.
This description immediately springs to our mind that one of the main purposes and strengths of this tool is the ability it gives us to debug the system at runtime.
One of the reasons we may need such an instrument in fact is very easy to appreciate, and we can show it directly through an example.
Let's assume we want to test a new rule that says:

"If there is a person with high blood pressure in the scene => then do this".

Surely is inconceivable to think that we should wait for (or, even worse, to provoke) a subject to have high blood pressure in order to test this particular rule.
It is then obvious that the interactive Clips allows us to debug the robot and the system in such cases (and not only).
Furthermore, the main feature of this interface is to be declarative. As we said earlier, the main goal is to have a declarative interface which enables the average user to use the system easily.

The idea of iClips was born right after we decided to integrate Clips in the system. We had Clips, that is a rule interpreter. Since it is embedded in a `C#` application, we implemented the syntax driven editor: the *Interactive-Clips*. This one represents the environment we used to embed Clips (using CLIPSNet) in `C#`.
Since the interface for controlling the robot -InteractiveClips- is still in progress, we realized a graphic control -iClips- which internally encapsulates the functioning/behaviour of the interface on Clips side. In this way, when the rest

of the interface will be polished, it will be updated with no efforts. Plus, as you can see from Figure 3.3, the interface at the moment is a little bit raw. However, the goal of this project was just to make it work, not to make it look good.

At this point we used specific features of the .NET Reflection to build our own notations, which allow the system to have a kind of *domain specific language*. The Interactive is already part of the architecture. It is a library which shows a window that will be embedded in another system. In fact, the Interactive is a module too: for the rest of the world iClips cannot be distinguished from Clips.

After saying what iClips is, we also want to see what it does.
This tool has several capabilities and represents a basic tool for running several kind of tests. We already listed several reasons why we built such a window (interaction, debugging, observation, forcibly assert facts etc). We want now focus on its strengths, by listing them.

- *It acts like the Clips editor.*
  This library is responsible of doing several things. It owns Clips within it, it creates the `ModuleManager`, it creates the interaction windows and so on. Within this editor the user can act exactly as if he/she were in a Clips environment. However, not every situation requires the window to be displayed, so it must be able to appear and disappear, and that's why we implemented methods like `showWindow` and `hideWindow`.

- *It allows the dynamic loading of modules.*
  The iClips tool makes use of the .NET Reflection to allow the user, by using custom attributes [2.5.1], to load modules at runtime and do other debugging operations related to modules. This feature has been described in the previous paragraph [3.3.1]. So, if we want to test a new or a specific module, we can do it easily through the InteractiveClips window.

- *It allows Code Completion and Highlighting.*

Trough this interface we get an editor complete of IntelliSense and Syntax Highlighting, two basic features for a syntax driven editor. The goal is achieved thanks to the AvalonEdit tool [2.6], combined with a smart usage of custom attributes. We get a personalized IntelliSense (code completion) which deals with both native Clips code and custom methods we have exposed to Clips ourselves.
When typing in the upper window (the input window, Figure 3.3) we obtain hints on the word we are typing, even if the word does not correspond to a pre-implemented Clips method but we added it on our own. Additionally, when we expose methods to Clips through the `ClipsActionAttribute`, we add all the information we want to retrieve later on. In this way, the CodeCompletion tool can retrieve them and make them available while typing to help you use the functions properly even if you don't know them at all (e.g. method signature, expected parameters, a brief description and so on, just like common IDEs would do).

- *It lets the user call `C#` exposed methods from Clips.*
  It does this and much more. It allows the loading of properly labeled `C#` libraries, making two things possible at once: you can assert facts from `C#` to the expert system and you can call specific `ClipsAction`-annotated `C#` methods from iClips. The automated exposition of the actions in the Clips world (following the declarative paradigm) is implemented by using meta-programming and reflection.

- *It acts as a specific Debugger.*
  We've already been through this concept. The key point is that we wanted a graphic environment which would allow us to observe the robot behaviour.

- *It works as a Declarative tool.*
  We've already been through this concept, too. The key point to have such a declarative tool is constituted by the mentioned use of reflection.

After saying how it is and what it does, the next step is to focus on how the iClips Interface is built.

The main tool we used for this purpose is *AvalonEdit*, a free source code editing component. We decided not to use the most popular *Scintilla* editing component [Soub], because we considered it less intuitive and more complex compared to the one we chose.
Since we used the AvalonEdit Text Editor [Prob], we wanted to exploit its functionalities such as *CodeCompletion/IntelliSense*, *Syntax Highlighting* and so on, creating then a nice and suitable iClips editor for our system.
As we said in section [2.6], the editor can be used like a simple WPF `TextEditor`. We just replaced it inside the `ClipsEditor.xaml` code.

The first goal of the iClips editor is to provide *Syntax Highlighting*.
This feature is implemented by creating a `.xshd` file specifically meant for Clips syntax. We named it `CLIPSHighlighting.xshd`: we will see this file when showing the implementation phase in the next chapter. The file mostly contains rules and keywords and the related colors to be applied when matching them. We wrote it by examining the Clips syntax from samples and tutorials, and by looking at the other existing `xshd` files as starting points.
Another goal of the iClips editor is to have the *Brace matching* feature implemented.
We used the `PortionColorizer` class for that purpose. In this way we obtained a syntax highlighting feature that highlights matching sets of braces, as desired. The purpose of brace matching is to help the programmer navigate through the code and spot any improper matching, which would cause the program to not compile or malfunction. Brace matching is particularly useful when many nested constructs are involved, and this is the main reason why it is such a useful feature in a language like Clips [Figure 3.4].

At this point we implemented the CodeCompletion feature by using AvalonEdit classes and .NET Reflection as well. The main class used to accomplish this task derives from the AvalonEdit `CodeCompletion.ICompletionData` and, together with the main string to be added as a new word in the

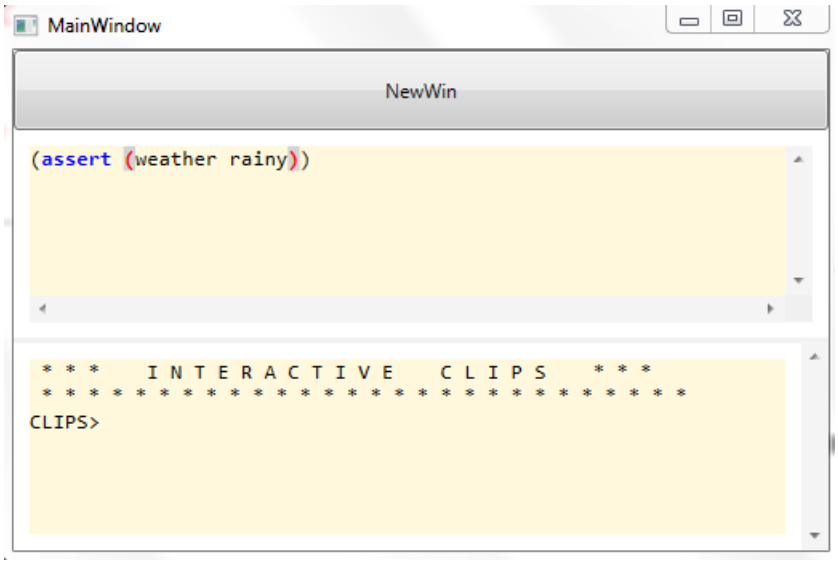

Figure 3.4: Brace Matching

completion window, it provides the possibility of adding features like priority, image, and description for the new entry.

The CodeCompletion feature has then been refined through the use of custom attributes. Let's see how.

When typing in the editor, CodeCompletion scans the database of types - through Reflection-, looking at the methods. For each of them it checks if the method is annotated with `[ClipsAction]` or not: if it is, a new entry in the CodeCompletion drop-down menu is created, and we can have it available together with the method signature and a shot description as well. That means, we worked in a way which allows data to be loaded via reflection.

Our goal, however, was not to have a complete database of every Clips method with parameters, signature, description and so on. It was instead to provide the future user with a proper structure, which should allow him/her to add this properties as desired/required.

Right now the CodeCompletion works with some Clips methods and keywords, plus ours annotated `[ClipsAction]` methods. We are still investigating whether a different data structure would be better for adding CodeCompletion words.

So, Reflection comes in iClips here in two flavors. It helps both with modules independence and loading, and with CodeCompletion. We basically used the ClipsActionAttribute Custom Attribute to have an easy way of annotating a method, in order to have it both exposed to Clips and available from the AvalonEdit CodeCompletion feature. Just remember the assumption we made on Clips Data types: ClipsAction labeled methods should only accept and return Clips data types.
In the next chapter the reader can find a code fragment related to the iClips Interface.

In summary, we can use the iClips interface as a normal Clips one. The only difference is that the iClips is more powerful for our purposes: from its editor you can call methods that allow you to load modules or execute conveniently exposed `C#` code through annotations.

3.4.1 A Graphic Control

After an initial development of iClips as an executable, we decided to switch to a graphic control.
Such a system in fact needs a dynamic environment in order to be switched across different windows and applications. This is the main reason why we decided to change our minds from the initially developed static editor into a new WPF graphic control.
If someone wishes to embed iClips, he/she needs an object which encapsulates the Clips environment plus its graphic interface: that means, he/she needs a library through which he/she can manage the melting pot of existing controls and graphic primitives exhibiting pre-defined behaviours.

By pursuing the goal of a graphic control, we found ourselves with the old `InteractiveClips` project turned into a graphic control, and a new `iClipsBox` one. Why do we need the `iClipsBox`? We designed it to clarify the difference between the graphic control and its container. In fact, the `InteractiveClips` represents our brand new graphic control and the `iClipsBox`

is its container.
Let's see the `InteractiveClips` first.
It is now built as a library and it contains primarily two `xaml` files: the `iClips.xaml` and the `ClipsEditor.xaml`.
This is due to the fact that we wanted to separate the concept of *"our window to be displayed"* and the `ClipsEditor` itself. We want our environment to be in the new `iClips` environment and the `ClipsEditor` to internally keep a reference to this `iClips` object.
Our goal is to have a graphic control which shows both a syntax driven editor (above) and the output window (below), as you can see in Figure 3.3. The `ClipsEditor` is a `xaml` user control, which uses an `embedEditor` tag to set -or check- if it belongs to a window. If you want the editor to be in your graphic control, you must set its `embedEditor` tag to `true`.
The question now is, why would we want an `iClips` graphic control separated from the `ClipsEditor`? We need it to be free to open and close the editor safely, without loosing the Clips internal state!
Now the `ClipsEditor` can be switched across windows.
In summary, `iClips` is a WPF graphic control that we use to embed Clips. It has no graphic interface, it just keeps a state. Then we designed a *"New Window"* button which opens a new window with the Clips editor when clicked. By clicking this button, the editor is moved into a new window. If you close this last one, it re-dock itself back to the initial window, always without loosing the state. Users can use this feature as they like.
So, now `InteractiveClips` is a dll which contains a graphic control used by `iClipsBox`. The `iClipsBox` is the container of the graphic control in our test application, and it displays the `ClipsEditor` at the moment. It can be viewed as our standalone executable, our test environment.

The new achievement enables the control to be moved from a window to another. From a conceptual point of view, we want to be sure that if at some point we want to re-instantiate the editor (e.g. if it gets messy) or instantiate more than one editor, we can do it. Since we don't know where this object is going to appear, we made it an auto-resize rectangle inside the available

space. The point is that the user may want this object as a little panel is his application, or together with another window maybe, or even showing up after clicking a button -and so on- and we must allow this possibilities to be carried out. By building iClips as a graphic control, we allow the system to have or not have a graphic interface, and by having it, to hide or display it.

3.5 YARP Module

The YARP Module can be viewed as part of the modules block. [Figure 3.2] As we already said in chapter 2, YARP is one of the main requirements of the system. Since the control system of the robot had to deal with local and remote modules, we decided to introduce YARP, a standard protocol for communication in robotics. Although we place it together with the other modules, the YARP module is not a simple module and because of that it deserves a separate discussion.

First of all, we should clarify what we mean by saying that YARP is *the communication mechanism* of the system. The YARP module is not required for the modules to *"talk"* with each other in general: the inter-module communication in fact occurs through the Clips expert system. Internal modules do not need YARP, they can assert directly into Clips. Those modules may (but to not have to) incorporate a module themselves to use YARP remote capabilities. YARP then comes into play when we talk about communication with modules/sensors which provide remote capabilities.
This is why a different approach is used for *"external"* modules, which must already have a YARP library, in order to be easy to integrate into the architecture.
To better understand the meaning of the `YarpModule`, it may make sense to consider a possible scenario. For example, if we have a sensor like the Kinect and we want to give it more CPU power in order to perform certain calculations, we would wish to move the sensor to a remote machine. At this point, however, the sensor can not remain isolated, on the contrary it will always be part of the architecture as a remote module.

Through YARP, we make sure that it can still talk to the *"brain"* (the Clips module) and be part of the architecture, still respecting the fact that the coordinator is only one.
At this point we do not care where this remote module is and how it is written, we just care that it uses YARP to talk with Clips, which then is able to take its messages and collect them.
If we want then to answer the question: *"Is the YARP protocol in our architecture a communication process between module?"* The right answer is *"It depends"*. It is the communication protocol between remote YARP nodes and YARP Modules, but it is never used for communication between modules (not even YARP Modules).

The `YarpModule` is a module that uses an external service to complete its tasks. It is a module that in turn encapsulates one or more modules that instead of running on the local machine are performed remotely. It acts like a proxy, lifting the developer of the module from having to write a YARP module within each system. It receives the request, posts a message, waits for the result and mails this result when it gets it.
So we took the YARP abstraction level that we consider to be reasonable (*e.g. the bottle*) and wrote a YARP module you can derive from. This abstraction level should provide guidelines, templates for sending/receiving, and so on.

This is the right moment to argue about pros and cons of using the Bottle structure. Let's say we want to use YARP to communicate generic instances of objects.

In a first scenario, we use the `Bottle` structure. That means, we first copy our instance fields into a bottle message, then we send it to the receiver. The bottle format is a standard to YARP, and this is an advantage itself. When sending the message, if it is serialized into a bottle, additional information such as types are carried together with their related values. Hence, the receiver is able to reassemble the message and decode it without having to know any other information. So, the receiver side is easier when using the

bottle.

In a second scenario, we don't use the `Bottle` structure, but a custom/specialized serialization instead (as we can see with the `"Target"` class example, section 2.4.1). We directly send the message through the network, saving a copy with respect to the bottle approach.
The more efficient side here is the sender side. There is another advantage: we can send through YARP our instance with no additional information: only data flows on the YARP connection. In this case, however, the receiver is not able to reassemble the message and decode it easily. It needs further information. This is the reason why we have to override the write and read methods as we saw in the previously mentioned example.
In summary, by using the `Bottle` structure we have less code to write, but more data passes through the YARP connection. Without using it, is the other way.

The YARP module in our architecture behaves following the Adapter Pattern [OODa], that means it behaves as an adapter whose job is to ensure that it makes no difference to the brain whether the assertions are local or remote, as we discussed.
A distributed robotic system needs such a tool: this is the main reason why YARP is a fundamental requirement of our architecture.
The way in which YARP ends up in our architecture however is not trivial and deserves further investigation. Our system was not specifying how to integrate YARP, but since it was a requirement we had to decide how to handle it. We decided to create a YARP module which is kind of our YARP base agent. The `YarpModule` is the skeleton module encapsulating the YARP communication. This means that the other agents will derive from the `YarpModule`, that is itself a class that derives from `Module`.
So the object model has a module class refined into two subspecies: one is the `YarpModule` in order to have a proxy and use YARP code, and the other is represented by the local modules (such as motor control, etc).
However, we must not be fooled by the `YarpModule` being a module of its own. It looks like this, but the concept is not that obvious. The module rep-

resents a technical part of course, but it extends beyond the simple module we have designed. If we had to answer the question *"Where is YARP in this architecture?"*, the right answer would be "everywhere".
YARP is implemented in the architecture through its own dedicated module, but this module is only a kind of template that ensures that when we will find ourselves with modules on different machines, they can communicate with the brain by using it. In addition, we wanted to guarantee that it was easy to access a single YARP connection. Facilitating the pattern is important: in this way you just need to configure as little as possible when adding new modules (e.g. having a generic/basic send and receive is a huge facilitation).

Now that we have been introducing a bit on the `YarpModule`, we can focus more on its specifications: how it is constructed, where it is in the architecture and what are the major challenges we faced in designing and implementing it.

The `YarpModule` is a `C#` class deriving from `Module` and encapsulating modules that, in turn, instead of being executed locally are executed elsewhere by YARP. When we will have new modules which use YARP to communicate, we will ensure that they derive from `YarpModule`, which in turn derives from `Module` (as similar to what we have done for Clips). These new modules then, in addition to having access to the shared Clips environment, will also share the YARP connector. The Clips environment and the YARP connector, are single instances: they both follow the *Singleton Pattern* [OODb].
By simply deriving from `YarpModule`, what we do in our "external" modules is hiding to the rest of the world the existence of YARP. External modules only need to know a predefined procedure when they want to communicate with the system. By following it, they can talk without worrying about where the YARP instance is, who created it and so on: everything is encapsulated in the architecture.
The `YarpModule` is nothing but the skeleton of a module which encapsulates the YARP communication. Inside it we have access to the `YarpManager`

shared between all modules, following the same template as we did for generic modules and Clips. However, it makes sense to keep the aspects separated: with the right dependencies it does not make sense that the `YarpManager` and the `ModuleManager` stay together.
We have already seen that the `YarpModule` acts as a proxy. The functionality provided by the remote modules passes through the YARP module and serves in a way to trigger an assert to the expert system.

At this point we can see what we need to do when we want to add a module with remote capabilities that communicates using the YARP protocol. Let's say we have our own module. First we make sure that it derives from `YarpModule`. For each action message (that is an input to the module YARP) we define a method and annotate it simply by the `ClipsAction` custom attributes we already know, in order to have the method exposed in Clips. For each output message instead (sent from the module) a suitable assert is carried out. This is a simple and well-defined procedure which tells you what to do to easily integrate a module that uses YARP into the architecture.

Chapter 4

ARCHITECTURE IMPLEMENTATION

The goal of this chapter is to give an idea of how the system has been implemented.
To do this, we will see the main features of the architecture described in the previous chapters directly by listing the main parts of the relevant code together with brief descriptions of the key points. Consequently, this chapter will be a lot less discursive than the previous ones, and more practical and code-oriented.
We will assume that the reader has a clear understanding of the concepts described so far, so that we can highlight the various aspects we have already discussed. The sections in which we decided to split this chapter on the implementation are: the implementation of the Clips, the iClips interface implementation and some practical examples of the concepts of reflection which we have already discussed. In order to have an overview on the architecture and its main classes, class diagrams are shown in Figure 4.1 and Figure 4.2.

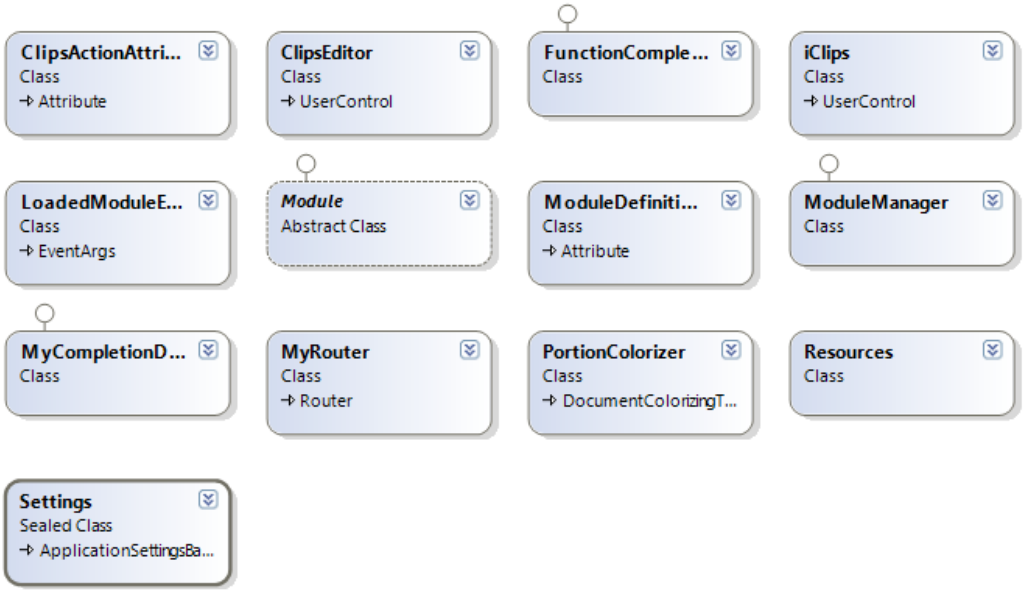

Figure 4.1: InteractiveCLIPS Class Diagram

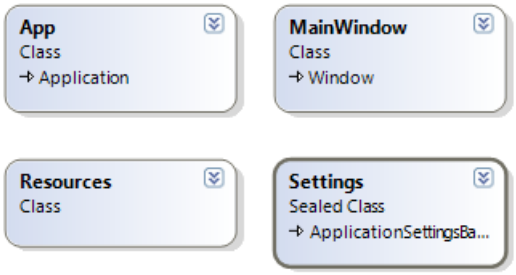

Figure 4.2: iClipsBox Class Diagram

4.1 A Generic Delegate

In the tools chapter we saw how it is possible to expose a `C#` function to Clips [2.1.4]. Since we evolved to a version of the system where a simple annotation obtained through custom attributes allows us to automatically expose the related method to Clips without doing anything else, let's see how this is possible.

Listing 4.1: Generic Create Delegate

```
1 public bool LoadModule(string name)
```

```
2  {
3    ...
4    foreach (var f in t.GetMethods())
5    {
6      var a = f.GetCustomAttributes(typeof(ClipsActionAttribute), true);
7      if (a.Length > 0)
8      {
9        var ca = a[0] as ClipsActionAttribute;
10       var pars = f.GetParameters();
11       var delegateType = GetFunc(pars.Length + 1);
12       var types = new Type[pars.Length + 1];
13       for (int i = 0; i < pars.Length; i++)
14       types[i] = pars[i].ParameterType;
15       types[pars.Length] = f.ReturnType;
16
17       var funcd = delegateType.MakeGenericType(types);
18       Delegate d = Delegate.CreateDelegate(funcd, m, f);
19       var clipsFun = new CLIPSNet.UserFunction(clipsEnv, d, ca.Name);
20       m.actions.Add(clipsFun);
21     }
22   }
23   ...
24 }
```

In Listing 4.1 we are inside the `ModuleManager` class.

This is the piece of code thanks to which we no longer need to create a specific `Delegate` whenever we want to expose a method to the rules word. It is important to remember that annotated methods to be used by Clips should use (both in taking and returning) only Clips data types.

The starting point is that the `CLIPSNet.UserFunction`, which links our `C#` function with the Clips one by naming it as we decided, wants the specific delegate for the method that should be created each time. We have then created a *"universal"* delegate, that is an n-ary one. Thanks to .NET Framework 4 we can handle up to 16 arguments. In this way, in the various modules we can write all the methods we want and just by labeling them, we will have them usable from Clips without having to write anything else. The piece of code we have just seen [Listing 4.1] handles the link between the two methods.

For each method that has been labeled (lines 4-6), it builds the particu-

lar instance we need (`funcd`, line 17). At this point we can use it for the `CreateDelegate` (line 18), and that's it.

4.2 core.clp and init.clp

Regarding the way in which we decided to use the system, we have seen that this has to be structured in a certain way.
The Interactive Clips application contains a Modules directory. Within Modules there should be two `.clp` files: `core.clp`, which contains the definitions of the core system (e.g. the basic definitions, `deftemplate` etc), and `init.clp`, which together are the main files of the `ModuleManager`.

Listing 4.2: core.clp snippet

```
1 ...
2 (deftemplate iclips-module (slot name))
3 (deffunction MAIN::require (?n))
4 (deffunction MAIN::require (?n) (return (if (not
5        (any-factp ((?module iclips-module)) (eq ?module:name ?n)))
6               then (load-module ?n) else TRUE)))
7 ...
```

What is the purpose of these files? They are the configuration of the system. For example, if we want the system to load modules at system startup, we just add in `init.clp` the command "`load-module moduleName`", or "`require moduleName`", which we shall see shortly. That's it.
`core.clp` in Modules however is a file that we do not want to be touched by users [Listing 4.2]. In it we define special meta functions that we want to be available in the system (such as the `require` method that we mentioned before), and we may define other functions related to the system and not with the individual modules. This is why we only expect expert users (programmers) to edit this the `core.clp` file within Modules. The `init.clp` instead, is used primarily for adding modules at the system startup, so it is less dangerous to modify.

Within Modules, in addition to these two files, there is a directory for each module of the system. Each module must have its own `init` file inside, which is not the same `init` described so far, but a file that we have already mentioned before (paragraph 3.3.1).
The modules are loaded separately. So when a module is loaded, its `init.clp` is called first. The file contains the initialization of each individual module such as its rules and functions, and can express "dependencies" through the *"require"* keyword.
What does it mean? If `moduleA` requires `moduleB` to be loaded, we can make sure that this happens by writing in the `init.clp` of `moduleA` the string "`require moduleB`." In this case, if `moduleB` was not already loaded when `moduleA` is loaded, "`load-module moduleB`" is called and `moduleB` is loaded, too; otherwise, if it was already loaded, nothing happens (it is not loaded again).
So when the system starts, the iClips module is the first to be started as an orchestrator, then it is in charge of loading modules.
By acting this way, we built a kind of configuration file that is in charge of loading all the modules that it requires. This file then represents the heart of the dynamic loading system of modules.
For each loaded module, the `ModuleManager` asserts the string:

```
clipsEnv.AssertString(string.Format("(iclips-module
    (name \"{0}\"))", name));
```

that informs Clips about that specific module being loaded. In this way we add this fact as knowledge to the rules world.
Let's examine more throughly the meaning of this *require* keyword. This keyword is very likely to be found in the `init.clp` file of every module. It is important to remember that *require* is a Clips function, not a fact. This is because if we had designed the require as a fact we would have asserted something which would only be triggered after the run of the Clips environment.
We need a function that loads the module exactly when we tell it to do it,

before continuing with anything else. This function is in the previously mentioned `core.clp` and is a good example to understand the meaning of this file. The `core` serves as a system configuration file for our architecture, so that we do not contaminate the `C#` code with definitions, Clips function calls, `deftemplate` etc.
Through *require* we solve the dependencies problem: if the module is not loaded, then it is loaded, otherwise, if it is already loaded, then it does nothing. The *require* function returns true if the loading succeeded or the required module was already loaded. Otherwise it returns false, so that we can then have something like `"if !require Kinect then error"`.

On iClips, thanks to the ***Deftemplate construct*** we described in paragraph 2.1.3.2 we can also assert facts regarding non-trivial structures.
This can be achieved as follows. Before facts are created, Clips must be informed of the list of valid slots for a given relation name. We can add this information in the `.clp` file of the module. In this way, groups of facts that share the same relation name and contain common information can be described using the same template.
Thanks to *deftemplates*, we are now able to write an `"assert"` that takes an array of objects as input. The main advantage is that, in addition to making sense in the Clips environment, deftemplates are useful and convenient when communicating with the .NET environment, too. In both cases, in fact, having classes or struct composed by fields is a well-accepted way to proceed (type checking is a related advantage).

4.3 Clips Routers

In paragraph 2.1.3.5 within the Tool chapter, we have already explained what the Routers are used for in expert systems. Briefly, routers are the mechanism used by CLIPS to handle I/O streams by specifying different sources and destinations. That means we have to re-define our own router and override its methods in order to get our Clips stream in the AvalonEdit editor.

Let's see how it works. We have our `MyRouter` class:

Listing 4.3: MyRouter.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InteractiveCLIPS
{
  class MyRouter : CLIPSNet.Router
  {
    private StringBuilder outputClips;
    public MyRouter(CLIPSNet.Environment env, string name,
        int priority)
      : base(env, name, priority)
    { outputClips = new StringBuilder(); }

    public StringBuilder Output { get { return outputClips; } }
    protected override bool Print(string name, string str)
    {
      if (str != "\n" && str.Trim() != string.Empty)
        outputClips.AppendLine();
      outputClips.Append(str);
      return true;
    }

    protected override bool Query(string name)
    {
      /*
      We want to intercept any I/O requests that the standard
      interface would handle. In addition, we also need to handle
      requests for the logical name top. The recognizer
      function for our router is defined below.
      */
      if ((name.CompareTo("stdin") == 0)
        || (name.CompareTo("stdout") == 0)
        || (name.CompareTo("wprompt") == 0)
        ||(name.CompareTo("wdisplay") == 0)
        || (name.CompareTo("wdialog") == 0)
        || (name.CompareTo("werror") == 0)
        || (name.CompareTo("wwarning") == 0)
```

```
40          || (name.CompareTo("wtrace") == 0)
41          || (name.CompareTo("wclips") == 0)
42          || (name.CompareTo("t") == 0)
43          || (name.CompareTo("RouterTest") == 0))
44        { return true; }
45        else
46        { return false; }
47      }
48
49      protected override int Getc(string name)
50      {
51        return base.Getc(name);
52      }
53
54      protected override void Exit(int exitCode)
55      {
56        base.Exit(exitCode);
57      }
58    }
59  }
```

Since we are dealing with .NET languages, we used the `StringBuilder` and the `append` method in order to accumulate text efficiently.
Then, inside `iClips.xaml.cs`, the router is declared and instantiated through:

```
1  ...
2   MyRouter router;
3  ...
4   router = new MyRouter(env, "RouterTest", 10);
```

And in `ClipsEditor.xaml.cs` we get the router output:

```
1  ...
2    private void executeSelection(string selection)
3    {
4      clipsEnv.Env.SetCommandString(selection + "\n");
5      bool b = clipsEnv.Env.ExecuteIfCommandComplete();
6      if (b)
7      {
8        testBlockOutput.Text = clipsEnv.Router.Output.ToString();
9        scrollBar.ScrollToBottom();
10     }
```

```
11   }
12 ...
13  router = new MyRouter(env, "RouterTest", 10);
14 ...
```

4.4 AvalonEdit meets iClips

In this section we will focus on the main aspects that are implemented in the AvalonEdit editor used by iClips. The behaviour of the editor and its basic features have been widely discussed in paragraph 3.4 of the Design chapter. The editor, as we already know, acts as a Clips window editor. This means that it emulates the Clips editor in which we can write Clips commands and obtain results and output. The main statement used to pass our commands (and in general whatever we write in the Avalon editor) to the Clips engine is:

```
clipsEnv.Env.SetCommandString(selection + "\n");
```

We want now to have a look to the salient snippets of code related to the highlighting, the brace matching and the code completion features.
Let's start from the Highlighting. It is mainly handled by the `.xshd` file that we specifically wrote for the Clips language. We can see the file in listing 4.4. Then we extended the SyntaxHighlighting feature of our editor by pointing it to the customized `.xshd` file.

Listing 4.4: CLIPSHighlighting.xshd snippet

```
1  <?xml version="1.0"?>
2  <SyntaxDefinition name="CLIPS"
3    xmlns="http://icsharpcode.net/sharpdevelop/syntaxdefinition/2008">
4    <Color name="Comment" foreground="Green" />
5    <Color name="String" foreground="Brown" />
6
7    <!-- This is the main ruleset. -->
8    <RuleSet>
9      <Span color="Comment" begin=";" />
10
```

```
    <Span color="String">
      <Begin>"</Begin>
      <End>"</End>
      <RuleSet>
        <!-- nested span for escape sequences -->
        <Span begin="\\" end="." />
      </RuleSet>
    </Span>

    <Keywords fontWeight="bold" foreground="Blue">
      <Word>else</Word>
      ...
      ...
      <!-- ... -->
    </Keywords>

    <Keywords fontWeight="bold" fontStyle="italic" foreground="Red">
      <Word>AvalonEdit</Word>
    </Keywords>

    <!-- Digits -->
    <Rule foreground="DarkBlue">
      \b0[xX][0-9a-fA-F]+ # hex number
      | \b
      ( \d+(\.[0-9]+)? #number with optional floating point
      | \.[0-9]+ #or just starting with floating point
      )
      ([eE][+-]?[0-9]+)? # optional exponent
    </Rule>
  </RuleSet>
</SyntaxDefinition>
```

The code completion feature is mainly handled by the `MyCompletionData` class. This class derives from `ICompletionData`, and implements it by defining some methods. The `MyCompletionData` class is very simple, so we will skip it.

We have already described how the completion works in paragraphs 2.6 and 3.4.

The code completion user interface is then implemented in the `ClipsEditor.xaml.cs` and `iClips.xaml.cs` classes. Here we manage the completion process by deciding e.g. when to open the drop-down completion menu (we

do it when the dot key is pressed), how to match words when typing in the completion window, and, more importantly, how to add to the completion data both from the functions declared in the `init` file of the modules and from our `C#` functions annotated with `[ClipsAction]`.

A code snippet of this feature is given in listing 4.5, from `iClips.xaml.cs`:

Listing 4.5: `iClips.xaml.cs` snippet

```
...
...
private void Init()
{
  env = new CLIPSNet.Environment();
  var dir = new System.IO.FileInfo((typeof(
    iClips)).Assembly.Location).DirectoryName;
  manager = new ModuleManager(env);
  manager.LoadedModule += (o, e) =>
  {
    foreach (var a in manager.LoadedModules[e.ModuleName].actions)
      editor.completionData.Add(new FunctionCompletionData
        (a.Name, "", a.Name, null));
  };
  new CLIPSNet.UserFunction(env, new Func<CLIPSNet.DataTypes.String,
    CLIPSNet.DataTypes.String,
    CLIPSNet.DataTypes.Boolean>((name, desc) => {
          var d = new FunctionCompletionData(desc.Value, null,
            name.Value, null);
          editor.completionData.Add(d);
          return true;
  }), "iclips-register-function");

  new CLIPSNet.UserFunction(env, new Func<CLIPSNet.DataTypes.String>
    (() => { var s = editor.openFileDialog(); return new
        CLIPSNet.DataTypes.String(s); }), "pick-file-name");

        int res = env.Load(dir + @"\Modules\core.clp");
        env.BatchStar(dir + @"\Modules\init.clp");
        router = new MyRouter(env, "RouterTest", 10);
}
```

```
32 ...
33 ...
```

In line 11 for each annotated `ClipsAction` method, we add its name to the completion data base. Lines 15-22 link the Clips `"iclips-register-function"` method to a function that takes a name and a description and adds the new function to the completion. Lines 24-26 link the Clips `"pick-file-name"` method to the opening of an `OpenDialog` box.
The behaviour of the `ClipsAction` annotation is shown by code in the next section.
So in the `init.clp` file we can also keep the list of the methods (by name and description) that we want to add as completion data. We will have several statements like:

```
(iclips-register-function "iclips-register-function" "Regis-
ters a function")
```

Even the completion list is interactively loaded from a file: in this way whenever we wish to add a keyword to the code completion tool, we only tamper the configuration file and not the code.

The last part of this section regards the brace matching.
The main class involved here is the `PortionColorizer`. It derives from `DocumentColorizingTransformer` and re-defines some methods such as `LineNumber`, `ColorizeLine` and `ApplyChanges`. Then it is used by the iClips editor. Here, when the user types, the `caretPositionChanged` event is triggered. This event calls a function which identifies the matching braces, whose offsets are used as parameters for a `PortionColorizer` object.

4.5 ClipsAction and ModuleDefinition

We have already seen which role reflection and custom attributes play in our architecture in paragraphs 2.5.1 and 3.4. We use this section to see the im-

plementation of the two main custom attributes of our architecture, helped by code snippets.

The first one enables us to annotate a method by using the `[ClipsAction]` custom attribute. By doing so, we get it both exposed into the Clips engine and added to the completion data as well. We do not have to worry of anything else: both our goals are automated with a single annotation.
Let's see the relevant code snippets. In the `ModuleManager`, we have:

Listing 4.6: ClipsActionAttribute definition

```
...
/*[AttributeUsage(AttributeTargets.Class |
        AttributeTargets.Constructor |
        AttributeTargets.Field |
        AttributeTargets.Method |
        AttributeTargets.Property,
        AllowMultiple = true)]*/
        public class ClipsActionAttribute : Attribute
        {
          public string Name;
          public ClipsActionAttribute(string name)
          {
            Name = name;
          }
        }
...
```

In this case the `AttributeTargets` are commented to show the reader how to assign targets. However, we did not assign any target, that means we get the default one: all targets.
Then, in the `LoadModule` function of the `ModuleManager` class, the methods annotated with `ClipsActionAttribute` are retrieved and added to be used from Clips. How? There is a procedure that retrieves all the methods of a class from the assembly: it finds, using reflection, all the methods which have been annotated with `[ClipsAction]`. It takes their signature, checks the arguments to be `CLIPSNet.DataTypes` and automatically generates the wrapper.

Instead, in the `init` method of the `iClips.xaml.cs`, as we saw in listing 4.5, the annotated methods are added to the code completion tool.
By acting this way, we obtain a sort of linguistic extension: the only thing the programmer should take care of is to label methods with `[ClipsAction]` at will.
An example of the usage of this custom attribute could be:

Listing 4.7: Custome Attributes Use Case

```
1  namespace TestModule
2  {
3    [ModuleDefinition(ClpFileName="init.clp")]
4    public class Class1 : Module
5    {
6      [ClipsAction("Add")]
7      public CLIPSNet.DataTypes.Integer test(
8          CLIPSNet.DataTypes.Integer i,
9          CLIPSNet.DataTypes.Integer j)
10     {
11       return new CLIPSNet.DataTypes.Integer(i.Value + j.Value);
12     }
13   }
14 }
```

Here we can see how easy it is now to write the functions that we want to expose to Clips. We just put the annotation on the method, taking care of using `CLIPSNet.DataTypes` for the arguments and return value. Now we can use the `"Add"` function (that is linked to the `C# test` method) directly from Clips; in our case from the iClips editor.

In this way, we really achieved one of the main goals of our work: *a declarative interface*.

The other custom attribute which help to obtain this result is the `ModuleDefinitionAttribute`.
We define it just as we did for the previously discussed attribute (see snippet 4.6). Now e.g. if we want to label a class with the `ModuleDefinition`

attribute, we act as we did for `Class1` in the 4.7 snippet.
Consequently, an instance of the `ModuleDefinitonAttribute` class with empty constructor is associated to `Class1`. We can add other information like `ClpFileName`, and this is equivalent to calling the `ClpFileName` setter property: the class metadata is now also labeled with this instance.
At this point, we can do:

```
typeof(MyModule).GetCustomAttribute(
        typeof(ModuleDefinitionAttribute));
```

Then we get back an array containing all the instances of this specific type which has been labeled that way. The main methods which are involved in this process are `LoadModule` and `FindModule`, in the `ModuleManager` class. `LoadModule` takes a string as parameter and returns a boolean. It then calls the `FindModule` method which in turn retrieves the module.
Inside the `FindModule`, we have:

```
var a = Assembly.LoadFrom(f.FullName);
```

We have it within a foreach construct. Since we are looking for a class which derives from Module, if the type derives from it then we are fine: we can actually load the module and return an instance of the class we found. This is the dynamic loading of modules: the `ModuleManager` implements it through reflection by using well-defined policies. We chose to have this mechanism implemented within the `ModuleManager` for various reasons. First of all, for debugging, then because the Interactive (that is an integral part of the architecture now) will eventually be embedded in other applications, finally because we want this mechanisms integrated in it.

4.6 YarpModule

Since the YARP protocol does not restrict the developer to a specific communication format, it is not possible to provide a generic way to abstract the

object communication. Hence the YarpModule is just a simple class whose unique goal is to abstract the YARP connection. Other choices, like the decision between using the `Bottle` format or not (we discussed pros and cons in section 3.5), are up to the developer.
YarpModule is nothing but an abstract class which ensures that the connection to the YARP reflector is unique and available to all the modules which specialize it.
The YarpModule subclasses can interact with Clips just like other modules do. From this point of view, we can see this module as a "normal" one with more functionalities: in addiction to Clips, it can also handle a YARP connection.

Modules that wish to use YARP can be very different from each others both in terms of languages and infrastructures.
We tried to implement a standard way to exchange data by using YARP. However, since we dealt with non-trivial data structures and generics, the main challenge was focused on the marshalling aspects. The implicit consequence is that we can't expect to have a standard way of passing data, because most YARP modules actually use *ad-hoc* representations of the objects.

Due to this reason, we decided to use `Managed C++` to write both the YarpModule and its subclasses.
In this case, `Managed C++` eases the communication in a .NET environment (e.g. CLIPSNet), and it can conveniently deal with pre-existing C++ modules. Using `Managed C++` gives a basic tool for people who want to use YARP, which is a good starting point both for them and for the rest of the system. This solution is the most complete which is possible without constraining the YARP communications. This is the main reason why we considered this decision both an acceptable and a sufficient solution.
This also explains why the SWIG automated approach did not work. The main reason is that marshalling is not simple to automate. As long as we deal with simple structures, the automated approach works fine. For more

complex cases (e.g. generics .NET), we need to manually handle the marshalling process.
For example, `C++` templates are not directly compatible with .NET generics because the underlying mechanisms are very different. This is the deep reason why we recommend the use of `Managed C++`. In fact, `Managed C++` can easily mix the managed and unmanaged world. That means we can use an unmanaged class from within a managed class, just like we actually did when testing some examples. This is possible because the interop between the managed and unmanaged worlds is carried out automatically by the compiler itself.
It is also possible to tune the system to fit specific requirements using the `System.Runtime.InteropServices` namespace, which provides support for interacting with COM interfaces, operating system calls, and general interoperability.

This aspect emphasizes that we can't expect to just have a single generic YARP module. Since the serialization of data when sending and receiving it over the YARP network is often *ad-hoc*, we will need a module for each specialized serialization pattern. On the other hand, we can provide all of the YARP modules with the guarantee that there is only a single YARP connection and that it is ready for communication.

Chapter 5

USE CASES AND TESTS

In this chapter we will describe some tests and use cases related to the work we have done.
The goal is to give a more practical view of what this architecture can be used for and how we can validate it. This is achieved by proposing examples that we actually implemented, but also use cases, together with achievable or just generic techniques.
We also want to provide the reader with suitable tools in order to understand how to use the system. The chapter is divided in two main sections: ***Tests***, in which we show some behaviours that we actually implement to validate the infrastructure, and ***Use Cases***, a second part in which we go over a series of examples and general approaches to the architecture. In this way we can show the advantages and the ways in which the user might decide to use the system.

To figure out what one can expect from this architecture, we focused on some examples which provide the rationale behind our choices. By doing so, we wished to find some situations in which a couple of rules were automatically able to do what it would have taken a lot of time to be written in a different way.
The main contribution of this thesis is having the architecture designed and validated; having the architecture already able to face everything it will even-

tually be able to do was not a priority. Then in the validation process there must be at least some expected behaviour (here comes the tests section) that, in addition of being valuable by itself, is also valuable in making us understand that it is now very easy to add other behaviours.

5.1 Tests

The starting point here is to make the reader aware of the fact that our main purpose is not having exhaustive behaviours implemented in the system, but rather to provide the infrastructure with the possibility of doing it eventually. So we carried out two main tests. These tests implement two specific, generalized and somehow exemplary behaviours.
The first one is related to the Kinect sensor and the second one is related to the FACE robot.

Let's start from the first one: **the Kinect Test.**
Its purpose is very easy and consists of making the Kinect sensor focusing on the subject face. As we already know from paragraph 2.3, the Kinect sensor is able to tilt up to 27° either up or down. We took advantage of this feature by exploiting the sensor as if it were acting much like the neck of FACE.
We sometimes used the Kinect instead of experiencing with the robot itself, because the robot was not physically available all of the time. This is possible because -depending on the experiment-, the sensor can fully emulate the same focus behaviours that we could have obtained from the same experiment carried out directly on the robot. From the test then, we obtained a sensor which is able to follow the subject's face up and down by using Clips rules in the new architecture.
It should be kept in mind that the tests we are talking about are not valuable by themselves but they are worthwhile to the validation process of the system: we never planned to implement the whole focus behaviour.

Let's see how it works.
The Kinect sensor by default acquires an RGB video stream using 8-bit VGA

resolution (640 x 480 pixels).
We decided that we want the robot to have the focus half-way along the vertical direction (240), but we don't want it to react to small movements so we defined a tolerance radius of 30 pixels. As long as the y-coordinate of the focused object is located within the 210-270 pixel range, we won't need any action from the sensor. Conversely, if the object turns out to be above or below this region, the sensor shall tilt upwards or downwards respectively.
We defined a `"faceCentre"` variable, which contains the y-coordinate corresponding to the computed face center of the focused object. So, whenever it changes, we assert it on Clips.
This test module exposes a single `moveKinect` method to Clips. We can see it in listing 5.1.

Listing 5.1: moveKinect method

```
1  [ClipsAction("moveKinectFun")]
2  public void moveKinect(CLIPSNet.DataType o)
3  {
4    if (o is CLIPSNet.DataTypes.GenericDataType<int>)
5    {
6      var s = (CLIPSNet.DataTypes.GenericDataType<int>)o;
7      try
8      {
9        if (((kinSensor.ElevationAngle + s.Value) <= 20) &&
10                ((kinSensor.ElevationAngle + s.Value) >= -20))
11         kinSensor.ElevationAngle = kinSensor.ElevationAngle+s.Value;
12     }
13     catch (Exception e) {Console.WriteLine(e.ToString()); }
14   }
15   else { Console.WriteLine(o.ToString()); }
16 }
```

Lines 9-10 check that the tilt angle is within the [-20°,20°] range because we don't want to break the limits of the sensor. (The device allows tilt angles up to 27° but we want to keep away from this limit).
So the sensor computes the `faceCentre` and asserts it together with the `middle` variable (the 240 half-way along the vertical direction) and our own fixed radius. In this way, whenever a Clips `"run"` occurs, suitable rules can

be fired. Our `.clp` rules and methods are:

Listing 5.2: TiltAngle rules

```
1  (deffunction MAIN::computeAngle (?p0 ?p1 ?p2))
2
3  (deffunction MAIN::computeAngle (?f ?m ?r)
4          (if (and (< ?f ?m) (> (- ?m ?f) ?r)) then (return 4))
5          (if (and (> ?f ?m) (> (- ?f ?m) ?r)) then (return -4))
6          (return 0)
7  )
8
9  (defrule MAIN::TiltAngle
10         ?factNum <- (faceScene ?faceCentre ?middle ?radius)
11         =>
12         (bind ?angle (computeAngle ?faceCentre ?middle ?radius))
13         (if (neq ?angle 0) then (moveKinectFun ?angle))
14         (retract ?factNum))
```

The main rule is the `TiltAngle` one, a simple rule which in turn calls the `computeAngle` function. If a fact like

```
(faceScene ?faceCentre ?middle ?radius)
```

is asserted, the `computeAngle` function is called to compute a *qualitative* sensor movement. We use the word *qualitative* to mean that the 4 and -4 values here are our arbitrary way to express *"a bit below"* and *"a bit above"*, respectively.
Through this function, the sensor is tilting and re-adapting itself incrementally, keeping then the focus on the subject.
Incremental focus in fact is a key point in robotics: it deals with the **Perception-Action Cycle** concept. The general idea of the incremental focus here is to move the sensor little by little, in a continuous re-adapting process. We will describe this aspect better in the next example.
At this point the method computes how much to tilt the sensor and the `moveKinectFun` is called. This last method executes `C#` code that has been exposed to Clips by using the `ClipsAction` attribute (listing 5.1, line 1).

The tilt angle is initialized to zero.

Let's now focus on the second test: **the Blinking Test**.
This one concerns a mixed active and involuntary behaviour on the FACE robot. We basically want to mix the following behaviours: on one hand, we want the robot to focus on a subject, following him/her by neck movements: this part, except for the coordinates conversion procedure in `C#` -that we will describe shortly-, is similar to the previous example. On the other hand, we provide an involuntary behaviour through the implementation of a rules set defining the blinking actions of the robot. This behaviour is then combined with the above intentional one.
Through this new set of rules, we are now able to show the *Clips rules priority mechanisms*. Let's see what we are talking about by better describing the blinking behaviour.
We built such a "situation" to have a qualifying example showing how a situation that was once difficult to manage is now easy by using expert systems and rules priorities.

Such a rule, if coded as `C#` rules together with hundreds of other rules might make the code unmanageable and/or unreadable. By using expert systems and rules priority, we have clean -and distinct- `.clp` files containing independent and orthogonal rules that are able to interact without having to know each other.
The blinking behaviour is built by defining some rules. The expected behaviour should be the following.
In order to simulate the human blinking, we want the FACE robot to blink every 15 seconds, for instance. This is the basic blinking behaviour, and it can be implemented by using a low priority rule representing the base-blinking cycle.
Then, as a further rule, we want to have the possibility for the robot to force itself not to blink (e.g. we can have it as a result of an "assert" from the iClips editor). At this point, we should have an even more prioritized rule that says: *"if you haven't blinked for 30 seconds, then blink!"*.

The point is that a human being blinks with some frequency, that ensures the eyes to stay humid. But you can voluntarily decide not to blink for a specific time-lapse. This action, however, can't last forever: at some point, human reflexes win.
In the meanwhile, every possible active behaviour can be carried out. This is why such a behaviour is so meaningful to observe and implement: it is a reactive behaviour which is able to interact with the deliberative behaviours without knowing how they are designed.
This turned out to be a good test because while it expresses a localized behaviour with a few rules, it allows the rest of the system to do what it wants. At the same time it shows that it is possible to build an otherwise complicated behaviour.

Let's see the details.
Since we wanted to take advantage of the Kinect sensor capabilities, we used it to run this test too. The Kinect sensor, however, is not placed in the robot eyes: that means when the Kinect sensor sees an object in a certain position, this position will not be the same as if it were computed by the FACE robot itself. As long as the Kinect captures objects/subjects positions, a new reference system must be computed to simulate the Kinect sensor being the robot's eyes. [Figure 5.1]
We assert to Clips facts like: `"faceAngle angle"`, where `angle` is the value of the angle between the x-axis of robot FACE and the focused object (e.g. `Teta2` in figure). In order to reach this goal, we faced some challenges.
We chose a polar reference system to express the differential focus with respect to the direction in which the robot is looking. Having such a reference system, however, requires two main goals to be reached. One is computing `"Teta2"` from `"Teta1"`, the other is the repeated coordinates conversion from cartesian to polar and back.
We aimed to build a generic system where the Kinect could be placed almost anywhere in the therapy room. So we designed a reference system in which we can set at the beginning the Kinect position with respect to the robot, then through a series of methods and conversions, the coordinates of the fo-

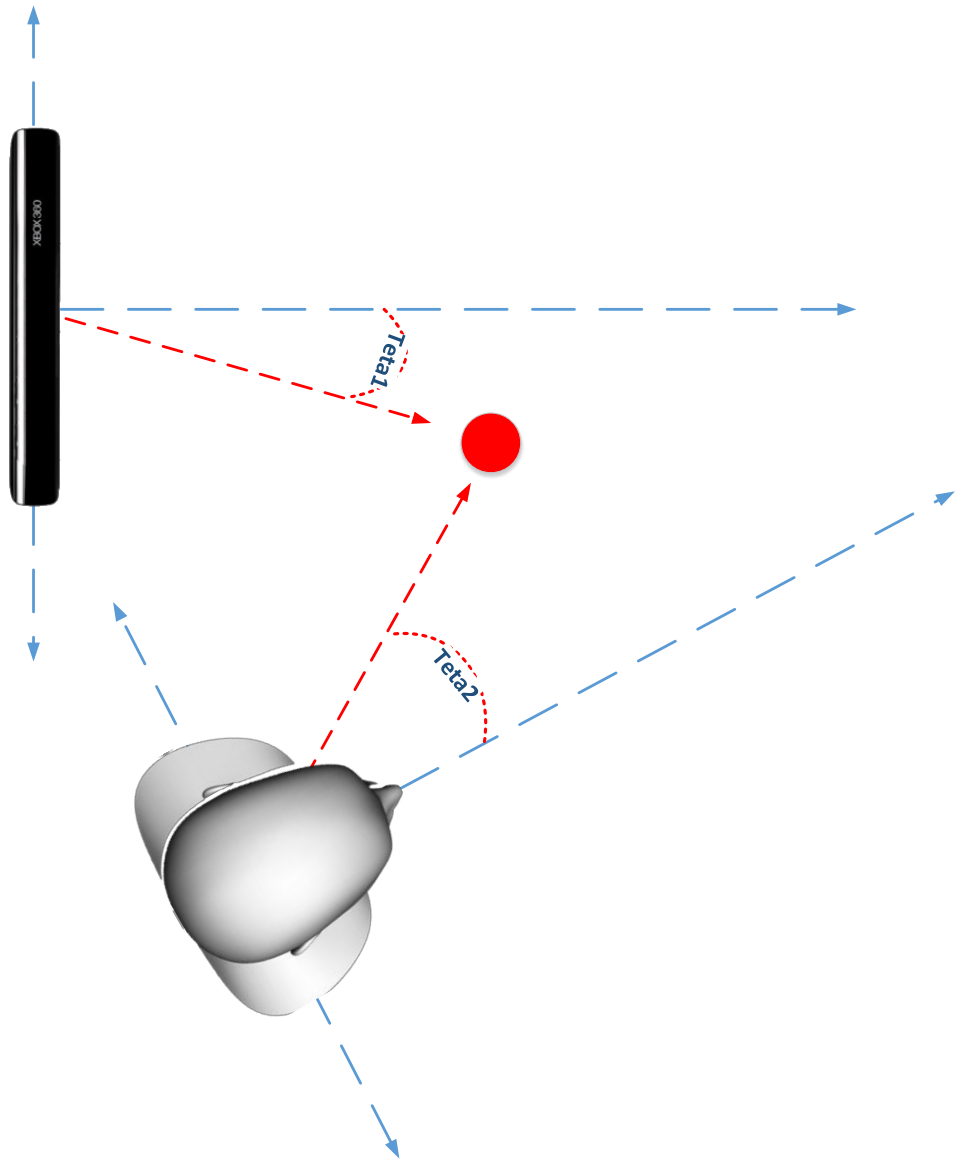

Figure 5.1: Reference Systems

cused object are computed as if they were observed by the robot itself.
We designed a reference space based on angles and distances, using then a polar space representation instead of a cartesian one.
All these information were collected to obtain *"qualitative"* assertions. That means, if the sensor knows e.g. that *"the subject is located at 30° with distance 0,5 meters"*, a fact like `"(turningFace alot right)"` is asserted.
We will see `C#` and Clips code snippets in a little. The qualitative notion of assertions that we have just mentioned is very important to such a system. We should never be able to assert something like `"turningFace 52° left"`, we must always work incrementally.

The main class that deals with coordinates conversion and change of reference system is the `ReferenceSystem` class. It computes the Joint Position coordinates with respect to the new reference system. Then, in the `Main` class, the Kinect sensor computes the Skeleton positions and its Joints.

```
...
Joint j = skeleton.Joints[JointType.Head];
if (j.TrackingState == JointTrackingState.Tracked) {
...
```

It first finds a tracked skeleton, if any, then it obtains its joints. We used the head joint. It is defined by a `SkeletonPoint` structure containing three floats as cartesian coordinates: `X, Y,` and `Z`. Then, thanks to the `ReferenceSystem` class and its methods, all the planned conversions are computed and we can assert the `"faceAngle angle"` fact, as we said before.
Regarding the focus part of the tested behaviour (the active behaviour of the test), we exposed to Clips the `C# turnFace` method.

Listing 5.3: turnFace method

```
1 [ClipsAction("turnFaceFun")]
2 public static void turnFace(CLIPSNet.DataType o)
3 {
```

```
  if (o is CLIPSNet.DataTypes.GenericDataType<int>) {
    var s = (CLIPSNet.DataTypes.GenericDataType<int>)o;
    try
    {
      float temp = currExpression.Face.ServoMotorsList[28].PulseWidth
                   + (float)s.Value * ReferenceSystem.turnShift;
      //Check Face turn angles max and min
      if ((temp >= 0) && (temp <= 1))
      {
        //update the Reference System
        float newangle = (float)(ReferenceSystem.refSystemRotationAngle
               * (180 / System.Math.PI)) + (float)s.Value;
        ReferenceSystem.refSystemRotationAngle = newangle *
               (float)(System.Math.PI / 180);
        //update Face turn motor (Channel 28)
        currExpression.Face.ServoMotorsList[28].PulseWidth = temp;
        currExpression =
          Body.ExecuteExpression(currExpression.Face, 10);
      }
      else { Console.WriteLine("Can't update Reference System!
                Turn Angle out of bound!!"); }
    }
    catch (Exception e) { Console.WriteLine(e.ToString()); }
  }
  else { Console.WriteLine(o.ToString()); }
}
```

The method basically translates the vector coordinates to servo motor movements in the FACE robot. Then we can see the most important code snippet of this test, the `.clp` snippet which defines the mixed focus/blinking rules.

Listing 5.4: FACE Focus/Blinking rules

```
(deffunction MAIN::computeAngle (?p0 ?p1))

(deffunction MAIN::qualitativeShift (?p0))

(deffunction MAIN::doBlink ())

(defglobal ?*lastBlink* = 0)

(deffunction MAIN::computeAngle (?h ?w)
       (if (and (eq ?h alittle) (eq ?w left)) then (return 2))
```

```
11        (if (and (eq ?h alittle) (eq ?w right)) then (return -2))
12        (if (and (eq ?h alot) (eq ?w left)) then (return 4))
13        (if (and (eq ?h alot) (eq ?w right)) then (return -4))
14        (return 0))
15
16  (deffunction MAIN::qualitativeShift (?f)
17        (if (and (>= ?f 3) (<= ?f 10)) then
18                        (assert (turningFace alittle left)))
19        (if (and (>= ?f -10) (<= ?f -3)) then
20                        (assert (turningFace alittle right)))
21        (if (> ?f 10) then (assert (turningFace alot left)))
22        (if (< ?f -10) then (assert (turningFace alot right))))
23
24  (defrule MAIN::HowMuchTurnFace
25        ?factNum <- (faceAngle ?fAngle)
26        =>
27        (qualitativeShift ?fAngle)
28        (retract ?factNum))
29
30  (defrule MAIN::TurnFace
31        ?factNum <- (turningFace ?howmuch ?where)
32        =>
33        (bind ?angle (computeAngle ?howmuch ?where))
34        (if (neq ?angle 0) then (turnFaceFun ?angle))
35        (retract ?factNum))
36
37  (deffunction MAIN::doBlink ()
38         (bind ?*lastBlink* (now))
39         (blink))
40
41  (defrule MAIN::Loop
42         ?factNum <- (blinkTrigger)
43         =>
44         (retract ?factNum)
45         (assert (wouldBlink))
46         (bind ?now (now))
47         (schedule "blinkTrigger" (+ ?now 15)))
48
49  (defrule MAIN::Normal
50         (declare (salience 0))
51         ?factNum <- (wouldBlink)
52         =>
53         (retract ?factNum)
```

```
54          (doBlink))
55
56  (defrule MAIN::Prevent
57          (declare (salience 10))
58          ?factNum <- (wouldBlink)
59          (notBlink)
60          =>
61          (retract ?factNum))
62
63  (defrule MAIN::Forced
64          (declare (salience 20))
65          ?factNum <- (wouldBlink)
66          =>
67          (bind ?now (now))
68          (if (>= (- ?now ?*lastBlink*) 30) then
69              (retract ?factNum)
70              (doBlink)))
```

`computeAngle` (lines 9-14) and `qualitativeShift` (lines 16-22) are Clips methods related to the focus behaviour. The rules that deal with this behaviour are `HowMuchTurnFace` (lines 24-28) and `TurnFace` (lines 30-35). The fact asserted from the module, `"(faceAngle ?fAngle)"` triggers the `HowMuchTurnFace` rule. This rule calls the Clips `qualitativeShift` method, that asserts facts like `"turningFace alittle/alot left/right"`.
In turn, these facts trigger the `TurnFace` rule, which calls the Clips `computeAngle` method first, and then (if the result is not zero) it calls the `C# turnFaceFun` function.

Then we have rules and methods dealing with the *blinking behaviour*.
The base-cycle blinking behaviour is expressed by the `doBlink` method plus the `Loop` and `Normal` rules. The possibility for the robot to force itself not to blink is expressed by the `Prevent` rule, which is given with higher priority at lines 56-61.
Then, as we mentioned before, this rule is eventually broken by an even more prioritized rule, the `Forced` one (lines 63-70).

Through all these rules we obtain the desired mixed active/involuntary

behaviour.
This `.clp` file, as we can see, hides everything we need to convert coordinates and change the reference system. It shows how we can abstract perceptions to obtain reference systems that turn out to be *qualitative*, designed and understandable. This was one of the main goals because it emphasizes that who is in charge of writing the rules does not need to know too many details about the underlying system.
In this case there is incremental focus, too. Like in the previous test, the main idea is always to have incremental adjustments of behaviour. This is translated to facts like `(turningFace alittle left)`. It should not be possible to have quantitative defined facts. So, whenever the attention needs to be quickly moved to a subject, we assert facts like `(turningFace alot left)`.

In both tests, then, we want a self-adjusting process.

5.2 Use Cases

In this section we go through the main possible *Use Cases*.
We have just tested some cases in the previous section. Our goal, however, is not to try every possible behaviour: our main goal is to validate the developed architecture.
We use this section to list a couple of cases which fulfill this purpose. These examples represent a mean to prove that the same goal is easier to reach when using the new architecture, instead of the previous one.
At this point, the system can be used as a starting point to design new modules and behaviours.

One of the worthwhile use cases is the possibility to alter the social behaviour of the robot. This can be achieved e.g. by temporarily interrupting the execution, asserting a new rule or loading a new module. These operations, in fact, can affect other pre-loaded behaviours unexpectedly, with the result of dynamically changing the robot behaviour on the fly.

That means, if we have a pre-loaded module and then we decide to add new rules or load new modules, then we can get even more unexpected behaviours, and this is certainly a good reason to have Clips as the main building block in the new architecture. Furthermore, the selective loading of different behaviours is potentially compositional. When adding or loading new rules and modules at runtime, we get a cartesian product of reactions as outcome, resulting in a complex behaviour. This contributes to enrich the FACE robot. So, combining different behaviours is a key point. Another one is the possibility to alter the system behaviour from the interactive window while the system continues working. Then we get debugging and modules independence.
Practical examples could be: pause the execution, force assertions, implement new behaviours and so on. e.g. you could pause the execution by asserting a `"debug"` fact, and even check the system status in that moment. Or, as we already said in paragraph 3.4 with the *blood pressure* example, you could decide to force a rule to be triggered by asserting specific facts. Furthermore, researchers and therapists can test selective focus and other behaviours.

Chapter 6

CONCLUSIONS

This thesis started from a need related to an existing robotic project: the FACE project.
We wanted to investigate the cognitive aspect of the robot, to provide it with an expert system, Clips, in order to allow the user to make the android *act like a human* and also interact with the system in several cases.
A second need that served as a starting point of this work was the desire to make the existing monolithic architecture as modular as possible, according with the most popular robotic systems requirements.
The thesis has concerned mostly the integration of the existing system with the Clips expert system, the YARP communication protocol, and the above mentioned code refactoring needed to turn the old architecture in a modular one.

An important contribution consists of the Interactive Clips, built as a declarative model allowing us to have a graphic tool to debug the system at any time. In fact, it is now our own window on the rules world. Through iClips, now we can perform even at runtime several operations that were just not possible before, such as the dynamic loading of modules and many more. The interface has a syntax-driven editor, which provides the user with code completion and syntax highlighting, specifically designed and developed for the Clips language and the functions we created.

Another contribution is the YARP communication protocol, which is used in the architecture as it is a standard protocol in robotics. Thanks to YARP, we are able to add remote modules using a standard platform for robotic applications which gives the possibility to expand our system over space and time without any problems. Robotic systems such as the one involved in fact, need more and more computing power over time. This is the reason why we can not expect to have everything on the same machine. In this case a standard protocol as YARP can be very useful for our purposes.
A further contribution of this work is given by the modular architecture that has emerged.

The proposed architecture allows the average user to intuitively and easily use the system, interact with it, etc., so that the *hybrid user*, that means a user without any special programming skills, can add new rules, check the status of the robot, dynamically add new modules and so on.
This is probably the goal that links together all the contributions that we have described so far. In fact, it makes it possible for users such as psychologists and therapists (which are key targets of the FACE project) to interact with the robot only with knowledge of the rules world, which can be obtained in a short time. It has been proved that simple *if then else* rules are easily designed by almost all kinds of users/subjects.
Among the major difficulties that we encountered are some problems related to the `C#` wrappers both for the YARP and for the Clips tools. In the first case we managed to overcome them without the use of the existing SWIG wrapper. Since we had to deal with non-trivial data structures, we decided to do the work ourselves. In the second case (Clips), we used the CLIPSNet library. Even in this case, however, we faced some problems with the types and we had to force the methods exposed to Clips to use the Clips data types. However, this represents the only constraint of the system.
Among the tools having particular relevance for the whole work we can mention the .NET Reflection and Custom Attributes. In fact, several features of the new architecture rely on them.
Use Cases and different examples are still on-going.

6.1 Future Developments

As often happens, the work we have been carrying out contains several aspects that we believe worthwhile to investigate or improve in the future. That means, various aspects may deserve further work.
One of these is the iClips interface itself. One of our goals was to give it a logical role, make it an integral part of architecture, and a key tool for modularization and refactoring. However, this does not mean that we focused on the purely graphical aspect of the interface itself. The result is, as we could see, a window which is functional and useful, but a little rough and not so nice to see. This can be one of the next steps.
Another aspect that may make sense to improve is represented by the code completion and its own set of words. In fact, we are still investigating whether it would be better to have a different data structure for adding new words in the drop-down menu.
Another step that we decided to accomplish when these small features will be added or filed, is to publish the iClips part of the project in order to provide users of projects other than FACE a tool to embed a Clips in their systems together with a full graphical interface. We wish to make the new iClips module available for the open-source community.
To conclude this section, we should mention the whole series of experiments and use cases that can still be tested on the system.

Chapter 7

ACKNOWLEDGMENTS

Ed eccoci qua, alla fine di questo percorso.
Meno di un anno fa questo obiettivo mi sembrava tanto lontano...invece questi mesi sono volati.
Non potevo però concludere senza ringraziare tutte le persone che hanno condiviso con me non solo questo lavoro, ma questo periodo in generale, fatto di traguardi, di scadenze, di momenti belli e altri difficili, ma comunque da ricordare.

Penso che comincerò proprio dal mio relatore, Antonio Cisternino, per avermi sempre spronato a dare il meglio e per aver creduto in me anche quando io stessa non l'ho fatto, bacchettandomi sistematicamente ad ogni *"non ce la farò mai"*.
Inoltre vorrei ringraziare il CVS lab, che stato la mia seconda casa in questo periodo e per questo mi mancherà. Penso di avere avuto una fortuna immensa nel potere lavorare in un ambiente così, fatto di persone straordinarie, di risate, di confronti, di torte, di scoperte, di impegno ma soprattutto di crescita, e non solo dal punto di vista informatico.
Ecco: magari il telefono che squilla ogni tre minuti e l'odore di olio...no, quelli non mi mancheranno. Forse.
In particolare vorrei ringraziare Andrea (di cui sono segretamente innamorata, Fox non me ne voglia), Leonardo e Simone. Per elencare i momenti "epici"

vissuti insieme tra laboratorio, mensa e macchinetta del caffè ci vorrebbe un altro capitolo, ma dalla regia mi dicono che non è il caso, quindi sorrido e vado avanti.
Un grazie anche al Morello, al Mura, Nicole, Maurizio Davini, il Cocco e tutti i componenti del lab.
Un ringraziamento di cuore, in tutti i sensi, va al mio Amore, Fox, che mi ha sostenuto facendomi sorridere anche nei giorni in cui ero intrattabile, aiutandomi, sopportandomi, spronandomi e viziandomi ad ogni occasione. Averlo accanto è stato un dono prezioso.
Ringrazio la mia famiglia e tutte le loro premure nei miei confronti, per essermi sempre stati vicini anche "da dietro le quinte", con il loro aiuto a volte tacito e altre esplicito.
Dedico a loro e a Fox questo traguardo perchè vederli così orgogliosi ed emozionati mi ripaga immediatamente di ogni singolo sforzo.
Infine ringrazio gli amici nuovi e quelli di sempre, e tutti coloro che in qualche modo hanno fatto parte di questo mio percorso, dalla mia Cecilia, a Leo, le sorelle Manfrè, Becciu, e tutti gli amici sparsi tra Pisa e Palermo, che sono stati direttamente o indirettamente parte di questa avventura.

Grazie a tutti!

Bibliography

[ACCE05] V. Ambriola, A. Cisternino, D. Colombo, and G. Ennas. Increasing decoupling in a framework for programming robots. *Dipartimento di Informatica, IMT Lucca*, 2005.

[AM94] R.C. Arkin and D.C. Mackenzie. Planning to behave: A hybrid deliberative/reactive robot control architecture for mobile manipulation. *Georgia Institute of Technology*, 1994.

[Cma] Cmake. Cmake. cross platform make. `http://www.cmake.org/`.

[Cse] Cse.wustl.edu. Ace. adaptive communication environment. `http://www.cse.wustl.edu/~schmidt/ACE.html`.

[Fei82] Edward A. Feigenbaum. Knowledge engeneering in the 1980s. *Dept. of Computer Science, Stanford University, CA*, 1982.

[FMN07] P. Fitzpatrick, G. Metta, and L. Natale. Towards long-lived robot genes. *Italian Institute of Technology, Genova*, May 2007.

[FMN12] P. Fitzpatrick, G. Metta, and L. Natale. *YARP User Manual*, 2012. online at "`http://yarp0/sourceforge.net`".

[Gia02] Joseph C. Giarratano. *CLIPS User's Guide*, March 2002.

[Gia06a] Joseph C. Giarratano. *CLIPS Advanced Programming Guide*, June 2006.

[Gia06b] Joseph C. Giarratano. *CLIPS Reference Manual*, June 2006.

[IET] IETF. Route reflector. `http://tools.ietf.org/html/rfc2796`.

[IFT] IFTTT. If this than that. https://ifttt.com/.

[JCG89] Gary Riley Joseph C. Giarratano. *Expert Systems, Principles and Programming*. PWS Publishing Company, Boston, 3rd edition edition, 1989.

[Laz09] Nicole Lazzeri. Face: A software infrastructure for controlling a robotic face for autism therapy. Master's thesis, Pisa University, 2009.

[Lib01] Jesse Liberty. *Programming C#*. O'Reilly Media, July 2001.

[MBL+10] D. Mazzei, L. Billeci, N. Lazzeri, A. Cisternino, and D. De Rossi. The face of autism. *RO-MAN, IEEE*, 2010.

[MFN05] G. Metta, P. Fitzpatrick, and L. Natale. Yarp: Yet another robot platform. *LIRA/Lab, University of Genova. MIT CSAIL, Cambridge*, 2005.

[Mica] Microsft. The .net framework. http://www.microsoft.com/net.

[Micb] Microsoft. Kinect for windows. http://www.microsoft.com/en-us/kinectforwindows/.

[MLZ+12] D. Mazzei, N. Lazzeri, A. Zaraki, A.D. Ahluwalia, and D. De Rossi. Facet: a human-robot interaction platform for emotional and social training. *GNB2012, Rome*, 2012.

[Mur00] Robin R. Murphy. *Introduction to AI Robotics*. A Bradford Book, November 2000.

[OODa] OODesign. The adapter pattern. http://www.oodesign.com/adapter-pattern.html.

[OODb] OODesign. The singleton pattern. http://www.oodesign.com/singleton-pattern.html.

[Proa] Code Project. The observer design pattern. `http://www.codeproject.com/Articles/47948/The-Observer-Design-Pattern`.

[Prob] The Code Project. Using avalonedit wpf text editor. `http://www.codeproject.com/Articles/42490/Using-AvalonEdit-WPF-Text-Editor`.

[QTG04] F. Qureshi, D. Terzopoulos, and R. Gillett. The cognitive controller: A hybrid, deliberative/reactive control architecture for autonomous robots. *IEA-AIE, 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert System*, 2004.

[Sch09] Robert J. Schalkoff. *Intelligent Systems. Principles, Paradigms, and Pragmatics*. Jones Bartlett Learning, November 2009.

[Soua] Sourceforge. Clipsnet. `http://sourceforge.net/projects/clipsnet/`.

[Soub] Sourceforge. Scintilla and scite. `http://www.scintilla.org/`.

[SR09] Peter Norvig Stuart Russell. *Artificial Intelligence: A Modern Approach*. Prentice Hall, PEARSON, 3rd edition edition, December 2009.

[Swi] Swig.org. Swig. simplified wrapper and interface generator. `http://www.swig.org/`.

[Wyg89] R. M. Wygant. Clips- a powerful development and delivery expert system tool. *Computers and Industrial Engineering*, 1989.