Increasing Decoupling in the Robotics4.NET Framework

Antonio Cisternino[1], Diego Colombo[2], Vincenzo Ambriola[1], and Marco Combetto[3]

[1] University of Pisa, Computer Science Department, Pisa, Italy {ambriola, cisterni}@di.unipi.it
[2] IMT Alti Studi Lucca, Lucca, Italy diego.colombo@imtlucca.it
[3] Microsoft Research ltd., Cambridge, United Kingdom, marcomb@microsoft.com

1 Introduction

The advent of social robots increases significantly the number and the kind of robotics systems to be controlled. Since CSRS software depends on the particular hardware architecture of a robot, software reuse becomes a significant issue. Reuse is usually achieved by packaging software in modules, by abstracting a well defined set of functionalities.

In the computer domain the variance of hardware has been addressed by abstracting common functionalities with drivers, software modules responsible for interacting with the hardware hiding the details to the rest of the system. In control software for robots [4] this approach presents several problems: first of all, it is very hard to provide an abstract description of robot functionalities independent from its structure. Moreover, the complexity of drivers increases because most of the activity performed by a CSRS is hardware control, located at the lower levels of the system architecture. Last, but not least, driver architectures focus on the notion of interrupt: when an interrupt notification gets lost the whole system may fail because of a lack of robustness in the communication schema.

Recently there have been attempts to define programming frameworks for developing robotics control software like those presented in this book, as well as several others [Bru01] [LBDS03]. Worth of notice is the new Robotics Studio, a new framework announced by Microsoft at the end of June 2006 [RS06] and based on .NET.

In this chapter we introduce the Robotics4.NET framework, its design, and how it contributes to achieve a better organization of control software because of the structure it imposes to CSRS. It is important to keep in mind

[4] In the rest of the chapter we will refer to this kind of systems as CSRS (Control Software for Robotics Systems).

that an overall design principle followed throughout the development of this framework is that the software is subdued to a well defined methodology and model. The framework is thus flexible enough within the constraints imposed by the model, avoiding helpful hacks that may break it; as a consequence of this design choice a programmer interested in its use is guided by the framework structure during the development, without having too many options for doing the same thing. We look positively at this design decision because an exceeding freedom in a framework often means that it does not capture the fundamental elements of an application domain, and the programmer will get little benefit by using it since the knowledge of the domain required to use it is significant. Moreover, the software developed under the proposed model (inspired by biology and not by software engineering) has interesting properties about possible reuse of software modules in different systems, even if based on completely different hardware.

At the end of the chapter we also briefly introduce the Microsoft Robotics Studio, and we compare it to our framework, looking for commonalties and differences, since both frameworks target the .NET execution environment.

2 Robotics4.NET

There are many different perspectives to look at the Robotics4.NET framework:

- the architecture it is inspired from a biological model in the hope that in the end the system will have similar properties;
- it provides a concurrent programming model based on message passing;
- it is a multi-agent system;
- it uses XML-based messages to allow heterogeneous architectures, allowing even micro-controlled systems to participate directly in the communication schema;
- it relies on customizable reflection support in order to declare communications aspects in the system;

All these aspects partecipate into defining a methodology that drives a programmer in a set of well defined steps in the software design process. In this section we discuss these perspectives, and finally we depict the overall methodology supporting framework's users in the development process.

2.1 Body Is First Class

The model behind the Robotics4.NET framework revolves around the notion of *body*: it encourages at organizing the software by defining a software counterpart of the physical body responsible to mediate the communication between the “brain” (i.e. the software responsible for control) and the body.

The choice of relying on the body is rather peculiar, if compared with other frameworks that focus on a separation of software in terms of components and a more classical functional decomposition of the problem like MARIE discussed in this book and [Bru01]. Nevertheless this choice has been driven by recent studies on the role of the body in the cognitive process of human beings [Hol04].

In the past two decades the notion of body has become relevant in the research on AI. Brooks [Bro85] [Bro91] [BS93] was one of the first promoters of embodied intelligence, with his hypothesis on complex behavior induced by a complex world. More recently neurophysiologists investigated the relation between the mind and the body and the environment [Dam94] [Dam99] [Dam03] and found that the body is tightly coupled to emotions and rational behavior. Nowadays embodied artificial intelligence [IPSK03] is a multidisciplinary research field whose aim is to study intelligence from a different perspective than the computational approach of traditional AI.

The body plays two fundamental roles in the human architecture:

- It is a controlled environment for the brain which perceives the world through it
- It provides a communication infrastructure to the brain such that a continuous stream of information about its state flows towards the brain

Organs are responsible for processing information acquired through senses and send the result of this activity to the brain for further processing (though there is also an information stream in the opposite direction [MR81]). Thus data coming from sensors is preprocessed before reaching the sensorial cortex.

The body seems to play also an important role in the self-consciousness of humans: as noted in [Dam99] and [Chu02] consciousness depends on the ability of the brain of distinguish the inner world from the outer world representations. This ability seems to be closely related to the continuous communication between the brain the body; besides, computers tend to prefer the notion of interrupt.

2.2 Execution Model

CSRS software based on the Robotics4.NET [CCEP05] framework has the following elements (see Figure 1):

- the brain: a software module responsible for the cognitive reasoning of the robot;
- the bodymap: a sort of black-board used to allow the brain to communicate with the rest of the system;
- a set of roblets: a roblet is a software agent communicating with the brain through the bodymap.

The brain and the roblets execute in parallel and communicate using messages. Roblets write messages into the bodymap; these messages are read by

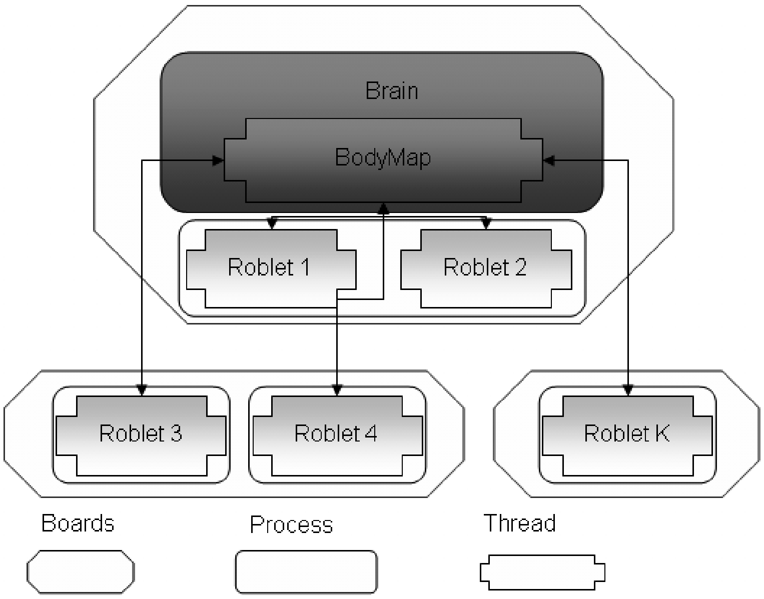

Fig. 1. The Robotics4.NET framework architecture.

the brain that in turn can send messages to them. Although the brain usually mediates communication among roblets, sometimes reactive behaviors require a direct communication between roblets. For this reason it is allowed a declaration of friendship between a pair of roblets that can communicate directly. This relation is asymmetric: a roblet interested in communicating with another one should declare friendship, though the destination of the communication is unaware of this relation.

Roblet execution is temporized: during the initialization phase, a roblet declares the frequency at which its behavior should be executed. A roblet gets notification of incoming messages from bodymap or roblet in friendship relation[5]. The brain has a more traditional organization and is responsible for handling its own control flow; for hosting and controlling the bodymap. The bodymap is also responsible for notifying arrival of incoming messages.

Systems based on the driver model optimize communications between drivers and the rest of the system: interrupts signal when something in the peripheral controlled by a driver changes. However, this communication schema is not robust: if for any reason an interrupt gets lost the entire system may fail. Think for instance to what happens if a software module fails in communicating that some end-run has been reached and a motor should be stopped.

[5] From a roblet standpoint the source of a message, brain or another roblet, is not relevant. However, it is possible to establish the source of the message.

In Robotics4.NET we are interested in robustness; therefore messages are sent asynchronously and the sender does not get any notification of message delivery. In this way, when a module fails the rest of the system continues to run; it is also possible to recover from the failure of a module simply by restarting it. Lost messages are not an issue in this context: roblets continuously communicates with the brain, thus if a message gets lost, there will be a next one bearing the information needed. In fact, roblets communicate continuously with the brain, whether there are changes or not, by pushing the status, in the very similar way as body organs always report to the brain; thus information exchanged is redundant ensuring that the loss of any message is not critical to the system.

The adoption of a message-oriented communication schema implies that there is no state of the connection to be preserved, and there is no acknowledge for received messages. Therefore, roblets may send messages without the bodymap or vice-versa. If a component goes down it can be restarted without affecting the rest of the systems.

The Robotics4.NET framework supports the development of CSRS software by providing suitable programming abstractions for the constituting parts. The communication infrastructure is declared in the program and automatically provided by the framework that gathers the required information from meta-data using the reflection API. Meta-data information is used to annotate types to let the framework know what relation exists among classes[6]. Thus any implementation of this execution model needs a runtime capable of providing reflection support.

2.3 Coordination

Roblets run in parallel and interact with the brain through the *bodymap*. They can also interact one with another either directly (using the *friendship* relation) or indirectly (when the a message from a roblet causes the brain to send a message to another one). It should be noted that, though the direct communication among roblets is possible, we highly discourage the use of this kind of communications since it affects the possible reuse of a roblet in different contextes. Thus the main communication structure of the framework is based on the roblets interacting through the bodymap.

It is natural to wonder how the communication schema proposed by our framework relates to classic coordination models. A roblet can send messages using two different *send* primitives:

- *Send(msg)*: used to send a message to the bodymap;

[6] Annotations in meta-data required by the execution model are limited to expressing relations among types. These can be expressed either implicitly, by defining types containing other types, or explicitly if it is allowed to store custom-annotations together with meta-data (custom annotations are supported both in .NET and Java since version 1.5).

- *Send(roblet, msg)*: send *msg* to the given *roblet*;

the framework is in charge for implementing the effective communication, avoiding explicit location references in the communication primitives. If we assume that the most relevant part of the communications is based on the first primitive, the communication schema closely recall that of blackboard systems of AI [Cor03] or the Linda coordination language [Gel85]. In our case, however, the communication through the bodymap is mediated by the brain software introducing a computation element not present in these models. Robotics4.NET can be placed in the middle between multi-agent systems and blackboard systems, retaining the nature of the first and the communication schema of the latter.

If the communication schema recalls blackboard systems, the multi-agent nature of the framework arises when we come to message delivery. There is no *receive* operation: roblets get notified of incoming messages through a callback. If a roblet is interested in waiting for a message, the main thread of the roblet should be suspended and resumed by the notification callback. This, however, is not a common pattern in roblets, and should be avoided if possible: the robustness of the framework is based on continuous communication that would be paused in this case.

Although it may seem that direct communication among roblets bypass the bodymap, this is true only in part. The bodymap is still responsbile for communicating delivery information about roblets to all the roblets that have a *friendship* relation with them. Yet, once this information has been exchanged, roblet can communicate directly. With this kind of interaction the framework gets closer to a multi-agent system in which roblets communicate through messages. Although this can be considered a more expressive and flexible framework, for the considerations about the role of the body, we believe that this aspect of the framework should be restricted to model of fully reactive behaviors in the robot.

In its first version the communication infrastructure provided by the framework was very simple. As it is discussed in section 2.4 we used the extensible meta-data provided by .NET to allow developers to declare the communication schema as part of the roblet definition. In the natural evolution of the framework we have found that the ability of controlling messages is very valuable and we are currently working to introduce simple message routing in the declarative communication model. We devised the notion of communication state of a roblet, and message delivery can be conditioned by it.

2.4 Implementation

If a contribution of Robotics4.NET is a methodology for designing CSRS software, its implementation is also worth of notice. It relies on the services provided by state-of-the-art virtual machines: dynamic loading of types and the ability of reflect the code structure at runtime. Our prototype runs on the .NET Framework[.NEb], as well as on Mono[Mon], and .NET Compact Framework[.NEa] and it has been developed using the C# programming language[HWG03].

The reasons for choosing the .NET framework are:

- Interoperability: Platform Invoke allows to easily map standard DLLs functions into class methods allowing to develop the lower levels of the architecture in languages such as C or C++;
- Networking: often the software is distributed and facilities help software components to communicate across processes or even computers is useful;
- Dynamic loading: the ability of dynamically load software modules is really useful in robotics, it can be used to load behaviors at need;
- Memory management: although the lower levels of the system are usually implemented in C++, algorithms developed for planning, unification, theorem proving, and similar tasks, greatly benefit from automatic memory management (and traditionally are implemented in programming languages with such a feature);
- Multiple languages: the Common Language Runtime is targeted by compilers of several languages (other than C#, Visual Basic and C++), ranging from Python, SML, Scheme, OCaml; this may help reusing existing software developed in the vast AI field.

In our experience, CSRSs are complex enough to require many of the services provided by virtual machines such as CLR and JVM. Often C++ framework have to implement (or link libraries) providing them, as it happens in the OROCOS system [Bru01]. Moreover the computational power is making possible to run shrunken versions of these runtimes on micro-controllers, making viable to program a robot as a highly distributed system.

In our implementation messages are represented by XML tree structures, obtained through XML serialization, and collected by the bodymap in a larger tree containing all the messages received from the body. In this way the entire status of the body is represented by a single tree, to the advantage of the brain that has a homogeneous data structure to deal with.

The adoption of XML as a wire format for messages has the advantage that even micro-controlled systems can partecipate actively as roblets, though they have to be programmed raw, without any support provided by the framework. We are currently using a micro-controlled board in the a robot that implements a roblet interface for sensors. A meta-program generates the C code hndling newtorking communications reading a .NET annotated set of types describing the messages.

The first time the bodymap receives a message from a roblet, it generates a special message requesting the roblet interface. When the message is received by the roblet the framework automatically generates a description of its interface based on the annotations present on the class definition. An interface consists of:

- an XSD schema (W3C Specification - XML Schemas) describing the input and output message types;
- for each message kind an instance provided as an example to be used by the brain;
- a description of the friendship relations.

The bodymap is responsible for receiving XML messages from running roblets and storing them into the message tree.

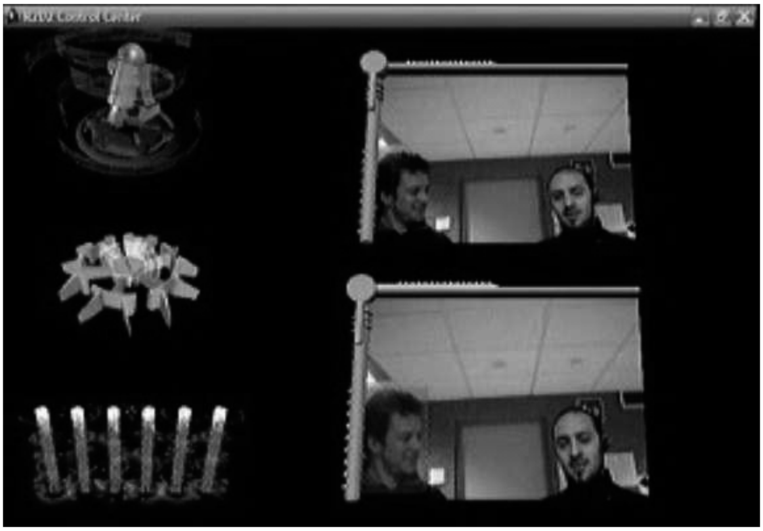

Fig. 2. A monitor of a robot implemented as a roblet.

We use UDP as a transport protocol, since we do not rely on an interrupt based model it is not a problem if some message gets lost. Besides, the stateless nature of communication helps to recover from modules or communications failures. UDP is also faster than TCP: we used standard XML messages to generate a video stream from a camera as shown in Figure 2. A drawback of using UDP based communications is that the maximum message size is bound by the particular MTU adopted by the local netework, as well as the network buffer used by the communication stack. In our experience we have found that messages are small enough to be sent over UDP, including those carrying the images in the monitor roblet in a standard ethernet based network.

2.5 A Simple Example

Let us consider the simple example of the *Heartbeat* roblet, which notifies the bodymap that it is alive:

```
public class HeartBeatMessage : RobletMessage {
 public long beat;
 public HeartBeatMessage() {
    beat = DateTime.Now.Ticks;
 }
}

public class SpeedUpBeatMsg : RobletMessage {
   public double freqChange;
}

[OutputMessage(typeof(HeartBeatMessage)),
 InputMessage(typeof(SpeedUpBeatMessage))]
public class HeartBeat : RobletBase {
  public override void Initialize()
  {
    OnSignal +=
      new OnSignalHandler(HeartBeat_Signal);
    Frequency = 0.5;
  }
  public override void Run() {
    SendMessage(new HeartBeatMessage());
  }
  void HeartBeat_Signal(RobletMessage msg) {
    if (msg is SpeedUpBeatMsg) {
      SpeedUpBeatMsg m = (SpeedUpBeatMsg)msg;
      Frequency += m.freqChange;
    }
  }
}
```

The *HeartBeat* roblet sends a heartbeat message every 30 seconds; its type is defined as a standard .NET Class, which inherits from *RobletMessage*, and it will be serialized into XML by the framework inside the method *SendMessage*. We use the UDP/IP protocol for sending serialized messages.

The *HeartBeat* class inherits from *RobletBase* all the behavior required to run. It is worth to note that we use custom attributes (the annotation on the *HeartBeat* class) as a mean to declare input and output messages of a given roblet. In this example we state that *HeartBeatMessage* is an outgoing message of the *HeartBeat* roblet and that the *SpeedUpBeatMsg* is an incoming one. Friendship is expressed through the *Friend* annotation that works in the same way of *InputMessage* and *OutputMessage* annotations.

XML serialization provided by the .NET framework is used to automatically generate XML messages from objects. The serialization process generating the messages is driven by the information stored in the program meta-data and accessed through reflection API.

Incoming messages are received by the framework, deserialized and notified to the class by calling the *HeartBeat_Signal* delegate. In our example the frequency of the heartbeat signal is changed.

2.6 Designing Robotics4.NET Applications

Now that we have introduced the framework, and discussed its design and implementation, we are ready to take a look to its typical use. A CSRS based on Robotics4.NET can be developed following these steps:

- body design;
- message definition;
- communication definition;
- roblet and brain implementation.

Body Design

In the first step the body is defined as a set of organs, each of them defined by a roblet. In this phase the granularity of the system is decided, since we have to establish the functionality of each roblet. A typical system has a vision roblet, one or more motion roblets, and one or more sensors roblets. The principal dilemma here is the way we group sensors and actuators.

In a robot, for instance, we defined a single roblet for all the sensors other than the camera. We also defined a motion roblet responsible for controlling robot movement; a head roblet, responsible for controlling head movements; and a body roblet which controls actuators used to change shape and internal sensors like end-run and batteries voltage detectors.

This process of grouping sensors and actuators of a robot is very important, and has influence over the system and on the ability of reuse software across different systems.

Message Definition

After laying out the overall architecture of the body, we proceed to define the nature of messages exchanged within the system. Although we consider it a separate step in the process, usually messages are defined incrementally, as the system gets refined during the development process.

The definition of messages is a typical problem of data definition, and several considerations can be inspired from those used in systems or database design.

Communication Definition

Most of the communication inside a typical body happen between the bodymap and the roblets. Friendship among roblets, however, are often required in real robots: they are often required to obtain timely delivery among organs, without giving the brain the opportunity of mediate the communication. These reflexes are also present in the human body and are critical to guarantee the most basic istincts such as collision avoidance.

Although this phase does not requires a significative amount of time, it is very important, since introducing friendship among roblets reduces the possibility of reusing roblets in different systems.

Roblet and Brain Implementation

This is the phase where we have to fill all the details in the system. Here we have to take several critical decisions for each elemtent of the architecture.

It is important to note that we can now focuses on the single aspects of the real system, having already addressed several difficult problems related to the overall architecture of it.

In this phase we also establish the timing of roblets: the frequency at which each roblet run (and typically send messages to the bodymap). These values highly depend on the nature of roblets and of the expected reactivity of the system. We believe that there is still room to better define this process, though for the moment we are still observing case studies in the hope of finding common patterns worth to abstract.

Final Considerations About the Design Process

As it can be noted the design process leave small room to the programmer for changing the basic assumptions. Although this may be perceived as a lack of flexibility of the framework, this is by design. We believe that frameworks are good when its clients can leverage on the expertise of the application domain experts that have designed them. If the framework provides only a set of unstructured features there will be more flexiblity in mixing them at the price of a greater knowledge from the client in their use.

3 Reusing by Means of Decoupling

Software reuse refers to the ability of a software unit to be used, unchanged, in different systems. Programming languages have developed many techniques and abstractions to support the creation of these units. Libraries are a popular way to isolate set of self-contained functionalities. Programming constructs like interfaces and classes are used to do the same at a smaller scale.

Software *coupling* refers to the amount of dependencies that a software unit has with other units. Some of these are unavoidable, like those to the standard libraries provided by the runtime support of a programming language. These dependencies, however, are not critical because language runtimes are required to run the program and widely available. Besides, remaining dependencies tend to influence the ability of reusing the software unit. As a rule of thumb the less a software unit is coupled the more is the chance to reuse it.

Robotics4.NET imposes a structure to CSRS software with a low coupling between roblets. A roblet is usually self-contained and has no other dependencies to other components; therefore it can be taken and employed as it is on another system. Whenever a roblet is friend to another one, we have a dependency that may reduce its reuse. However, as it has been pointed out in the previous section, the friendship relation is not symmetric, thus the target of any friendship has no dependencies from other roblets. Moreover the framework recommends avoiding the friendship relation unless strongly needed; this is because the brain is unaware of this kind of communication, losing control over the state of the body.

There is no dependency induced by the compilation of the framework: roblets and brain are not required to share classes (for instance those used to define messages). Message-based communication helps the development of heterogeneous systems.

Decoupling of roblets is what we call *horizontal decoupling* of the architecture: the brain orchestrates communication among modules that do not communicate directly. Robotics4.NET provides also *vertical decoupling*: roblets are a sort of drivers hiding hardware from the brain and defining the amount of abstraction of the body.

Let us consider, for instance, a roblet responsible for controlling a camera; in this case we have the option of sending to the brain the raw data read by the camera (this would be what a driver would do), or we can think of a roblet performing vision tasks and sending the results of the analysis to the brain in the form of a set of objects describing faces, objects, and whatever elements has been detected in the input stream.

Vertical decoupling is important because robotics systems tend to adopt similar hardware, though in different configurations. Therefore it is not infrequent that a roblet can be reused as it is on different robotics systems.

We have positively experienced the ease of reuse, by successfully reusing roblets in different robotics systems.

For instance, we developed a system to support sailors during regattas. The first version of the system has been developed as a distributed application with a Personal Computer reading the boat bus, and some PDAs providing information to people on-board. Afterwards we ported the application to our framework, getting a significantly better implementation. Although the boat is not considered a robot, it can be easily thought as if it is (this is possible for a large category of systems and devices), thus we had to define the body, so we decided how many roblets should have been developed and with which characteristics. During the re-engineering process the data flow of the framework helped us to spot a better software organization than that used in the first place obtained through a standard software design process.

We have been also able to successfully reuse two software modules on ER1, a robot based on the robotic kit by Evolution Robotics (Evolution Robotics Web Site), and on R2D2, an experimental robotic platform developed at our department[CCEP05] - whose appearance and functionality has been inspired from the popular Star Wars saga's droid.

In this case we have used ER1 as a smaller-scale model of the bigger and more complicated R2D2 robot. Roblets concerning interaction with the wireless communication, and camera processing, have been reused without modification. Most surprisingly we have been able to provide similar abstractions for motion and sensors roblets, though the two robots are significantly different in their structure.

An important aspect of the framework is the opportunity to group together different activities and code under the Roblet abstraction. The underlying metaphor of defining the body helps us to find, as it happens for organs, that a roblet may combine different tasks together because they are related to a specific organ of the robot. During the design phase it is reasonably natural to group the functions related to a specific hardware component together: this reduces communication and provides a suitable unit to be exposed toward the rest of the system. Thus we group close and meaningful functionalities together in a coherent module, which can be potentially replaced by another one exposing the same set of messages. With roblets we tend to reduce outbound communication because it is natural to keep information passing inside roblets.

4 Microsoft Robotics Studio

Robotics Studio [RS06] is a new framework for programming robots targeting the .NET execution environment. It is an interesting exercise to compare this framework with Robotics4.NET, both to compare choices made by different software architects and try to understand if there is a common ground shared by different frameworks.

Figure 3 shows the core architecture of Robotics Studio. The framework provides fundamentally three main services to application developers:

- Concurrent programming
- Type-oriented, message based communications in a service world
- Direct access to services through the *Decentralized System Services* (DSS)

Concurrency is an essential element of robotics applications, thus Robotics Studio provides for a concurrency model based on a library called *Concurrency and Coordination Runtime* (CCR). Goal of the CCR is to provide a concurrency layer that is more abstract than the standard concurrent programming

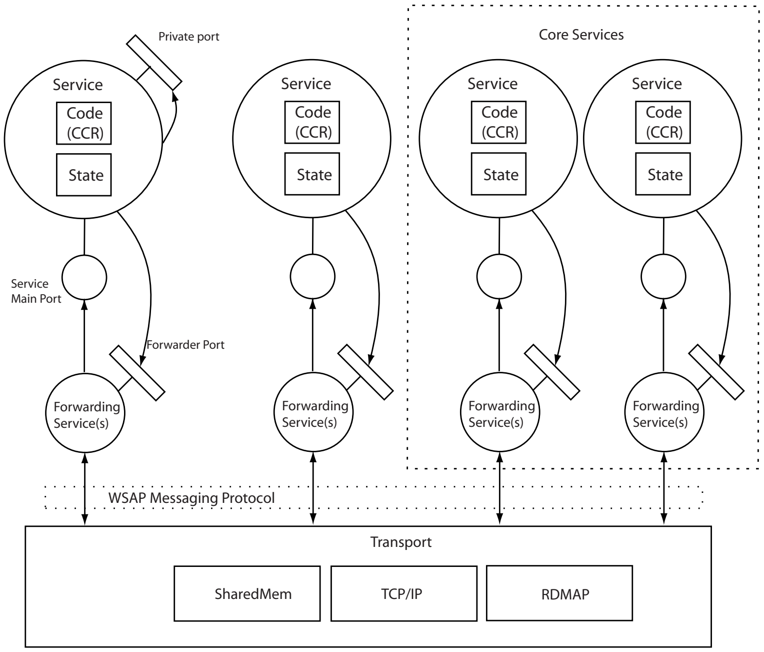

Fig. 3. Robotics Studio execution structure.

paradigm based on threads and synchronization objects. Concurrent activities (tasks) are managed by a dispatcher that is responsible for their execution. Communication is message based, and messages are enqueued into ports (see the figure) waiting for further processing. Coordination is achieved through *Arbiters*, classes responsible for implementing synchronization schemas based on ports. An example of arbiter is the *JoinedReceive* class that monitors two ports waiting for two input in order to execute a task. Tasks are provided in the form of delegates (objects that allow to invoke a method of an object through them), while the *Dispatcher* is responsible to schedule these tasks, possibly managing a thread pool. The following is an example of CCR based program [MSDN06], and shows how to perform an asynchronous read from a file:

```
private static void AsyncStreamDemo(DispatcherQueue dq) {
    FileStream fs = new FileStream(@"C:\Boot.ini",
        FileMode.Open, FileAccess.Read, FileShare.ReadWrite,
        8 * 1024, FileOptions.Asynchronous);

    Byte[] data = new Byte[10000];

    Port<Int32> bytesReadPort = null;
    Port<Exception> failurePort = null;
    ApmToCcrAdapters.Read(fs, data, 0, data.Length,
        ref bytesReadPort, ref failurePort);

    Arbiter.Activate(dq,
        Arbiter.Choice(
            Arbiter.Receive(false, bytesReadPort,
                delegate(Int32 bytesRead) {
                    Array.Resize(ref data, bytesRead);
                    Msg("Read completed, bytes read={0},
                        data follows:{1}{2}",
                        data.Length, Environment.NewLine,
                        Encoding.ASCII.GetString(data));
                }),
            Arbiter.Receive(false, failurePort,
               delegate(Exception e) {
                   Msg("Read failed, {0}", e.Message); })));

    HitEnter();
}
```

Typeful communication is another element of Robotics Studio; all messages have a type, and ports have a type (the parameter of the generic type *Port*). Support for types is provided by the .NET runtime, as well as reflection and serialization required to serialize messages as XML messages.

Services are based on CCR ports and tasks; each service thus is a set of concurrent activities with a state and a port to communicate. The state of a service is declared using custom attributes to annotate fields of the service class that are to be considered part of the state. The *DSS* infrastructure allows to access, modify and inspect the state of services from a Web browser.

Robotics Studio provides a model that enforces programming patterns that often arise in the development of software for controlling robots. The lightweight concurrent model provided by CCR is such that it is conceivable to define activities with a very fine grain, up to associate a concurrent activity with a sensor to be read. Types and reflection allow providing the communication infrastructure, as well as the Web interface allowing state inspection of a running system. It is important to notice, however, that the framework does not impose a structure to the control software: it is up to the programmer to decide how the concurrent activities are to be used to represent concepts. Besides, the framework is very flexible, allowing very different architectures to be represented in it.

It is interesting to note that Robotics4.NET and Robotics Studio share several elements in common:

- communication infrastructure automatically provided

- custom annotations used to let the programmer declare facts about the program
- concurrency is a fundamental building block of these kind of systems

Besides, the two frameworks also show several differences. The most important is that while Robotics4.NET aims at providing a rather constrained model and defines a methodology for programming robots, Robotics Studio gives a set of mechanisms that the programmer can use to implement the software. Another design choice that differs in the two cases is the granularity of the concurrency model: in Robotics Studio the model is very fine grained while in Robotics4.NET roblets provide a coarse-grain element for concurrency. Finally communication geometry is essentially flat in the first case while it is fixed in the latter.

From these consideration we can draw the following conclusions:

- Concurrency, reflection, and typed, message oriented communication are elements useful for programming robots.
- Virtual machines like .NET CLR are a good start to implement these services. Frameworks not reliying on this kind of execution environments tend to implement several services otherwise already present
- abstractions based on these core services can be very different, and there is no well established practice

As a final consideration, we believe that Robotics4.NET can be implemented on top of Robotics Studio. In this case we can retain the methodology aspects of the framework and the constraints that help the programmer in designing the software without having to maintain two sets of core services.

5 Conclusion

In this paper we presented Robotics4.NET, a framework for developing CSRS. The framework contribution is twofold: on one hand it provides automatically a communication infrastructure among the modules forming a CSRS system in a declarative way based on standard language independent mechanisms; on the other it provides a metaphor to guide programmers into software development. The project is still under development, though the first results are quite promising.

We have successfully reused roblets on very different systems, ranging from embedded systems based on Windows CE to full PCs, as it is. Because of the text-based format of messages exchanged by roblets, we have been able to integrate programs running on microcontrollers to gracefully integrate with the rest of the architecture even without running the framework due to hardware restrictions.

We believe that a major contribution of the framework is the choice of providing to the programmer a metaphor and a programming model that devises a well defined set of abstractions and an execution model. This is rather different from the approach taken by other frameworks tackling the same problem: they provide a set of functionalities together with full freedom in the organization of the modules and the communications among them.

We have also introduced the new framework for robotics announced by Microsoft. After a short presentation we compared the approaches of the two frameworks, comparison made interesting by the fact that both frameworks share the same execution environment; moreover, they tend to build on top of the same core mechanisms provided by it.

References

[Bro85] R.A. Brooks, *A robust control system for a mobile robot*, A.I. Memo 864, Massachusetts Institute of Technology Artificial Intelligence Laboratory (1985).

[Bro91] R.A. Brooks, *Intelligence without reason*, Proceedings of the 12th International Joint Conference on Artificial Intelligence (1991), 569–595.

[Bru01] H. Bruyninckx, *Open robot control software: The OROCOS project*, Proceedings of IEEE Int. Conf. Robotics and Automation (2001), 2523–2528.

[BS93] R.A. Brooks and L.A. Stein, *Building brains for bodies*, A.I. Memo 1439, Massachusetts Institute of Technology Artificial Intelligence Laboratory (1993).

[CCEP05] A. Cisternino, D. Colombo, G. Ennas, and D. Picciaia, *Robotics4.NET: Software body for controlling robots*, IEE Proceedings Software **152:5** (2005), 215–222.

[Chu02] P.S. Churchland, *Self-representation in nervous systems*, Science **296** (2002), 308–310.

[Cor03] Daniel D Corkill, *Collaborating Software: Blackboard and Multi-Agent Systems & the Future*, Proceedings of the International Lisp Conference (New York, New York), October 2003.

[Dam94] A.R. Damasio, *Descartes' error: Emotion, reason, and the human brain*, Avon Books, 1994.

[Dam99] A.R. Damasio, *The feeling of what happens: Body and emotion in the making of consciousness*, Harcourt Brace & Co., 1999.

[Dam03] A.R. Damasio, *Looking for spinoza: Joy, sorrow, and the feeling brain*, Harcourt Brace & Co., 2003.

[Gel85] D. Gelernter, *Generative communication in linda*, ACM Transactions on Programming Languages and Systems (TOPLAS) **7:1** (1985), 80–112.

[Hol04] O. Holland, *The future of embodied artificial intelligence: Machine consciousness?*, Lecture Notes in Artificial Intelligence, vol. 3139, pp. 37–53, 2004.

[HWG03] A. Hejlsberg, S. Wiltamuth, and P. Golde, *The C# programming language*, Addison Wesley, 2003.

[IPSK03] F. Iida, R. Pfeifer, L. Steels, and Y. Kuniyoshi, *Embodied artificial intelligence international seminar, Dagstuhl castle, Germany, july 7-11, 2003, revised papers*, Springer Verlag, 2003.

[LBDS03] T. Lefebvre, H. Bruyninckx, and J. De Schutter, *Polyhedral contact formation modeling and identification for autonomous compliant motion*, IEEE Transactions on Robotics and Automation **19:1** (2003), 26–41.

[Mon] *Mono project web site*, available at http://www.mono-project.com/.

[MR81] J.L. McClelland and D.E. Rumelhart, *An interactive activation model of con-text effects in letter perception: Part 1. an account of basic findings.*, Psychological Review **188** (1981), 375–407.

[.NEa] *.NET Compact Framework web site*, available at http://msdn.microsoft.com/smartclient/understanding/netcf/.

[.NEb] *.NET Framework web site*, available at http://msdn.microsoft.com/netframework/.

[RS06] *Microsoft Robotics Studio*, available at http://msdn.microsoft.com/robotics/, Accessed: 4/9/2006.

[MSDN06] Richter, J., *Concurrent Affairs*, available at http://msdn.microsoft.com/msdnmag/issues/06/09/ConcurrentAffairs, Accessed: 4/9/2006.