Abstract. Robots are becoming widely available and at affordable prices, however we do not have a suitable software infrastructure for programming them yet. A major issue in the field is software reuse: software for robots is inherently tied to the underlying hardware that is difficult to share among different platforms; nevertheless the software modules controlling hardware are often responsible for tough tasks so that a driver-based approach does not fit well the problem. We believe that the missing ability of reusing software in this particular field is due to the way software is organized following a functional decomposition of software modules. In this paper we present Robotics4.NET, a framework for programming robots that takes a different approach at modelling software modules in this domain. We discuss how our approach tends to achieve better decoupling and dependence of the modules in the system, thus making easier software reuse even on different robotic platforms.

Introduction

Robots are hitting the mass market, becoming not only elements of the industrial production system, but also active part of our lives. We call the last kind social robots: robotic systems sharing the same environment where humans live (houses, streets, building, etc.).

Social robots differ in many ways from their industrial counterpart: in an industrial setup robots should be correct under the assumption that humans conform to a well defined set of rules; moreover these systems should be optimal with respect to the production goals. Besides, social robots must adapt to the rules imposed by a human environment, being robust enough to be effective and plausible in their interaction with people. As a consequence, the control software of these systems can be very different in the two cases. Performance can be better achieved by increasing the number of assumptions whereas robustness is more about dealing with a large number of possibly not well known situations.

The advent of social robots increases substantially the number and typology of robotic systems to be controlled. Thus the ability of reusing control software for these systems is compelling. Recently there have been attempts to define programming
frameworks for developing robotic control software. Frameworks devise a methodology for developing software in a particular domain, helping the programmer by providing suitable programming abstractions. Moreover, software reuse is generally helped by the homogeneity imposed by the framework use.

In this paper, we briefly present Robotics4.NET [R4N], a framework designed to support software development for social robots. Then, we discuss how the framework design contributes to achieving a better organization of control software, due to the way software is organized. We finally briefly compare our design to those employed in different frameworks that follow a traditional approach to organize the software.

Robotics4.NET

Robotics4.NET is a programming framework aimed to support the development of control software for robotic systems. Its design promotes also a methodology of how to organize the software through a set of programming abstractions. The model has been inspired to the biological structure of the human body; this choice has been driven by recent studies on the role of the body in the cognitive process of human beings in the research area of Embodied AI [EAI].

A control system based on Robotics4.NET has the following elements:

- **The brain**: a software module responsible for the cognitive reasoning of the robot;
- **The bodymap**: a 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 run in parallel and communicate using messages. Roblets write messages into the bodymap; these messages are read by the brain that can send messages back to them. Although the brain generally mediates communication among roblets, sometimes reactive behaviors require a more direct communication model. For this reason, it is possible to declare friendship between a pair of roblets that can communicate directly. It is worth noting that apart from friendship (that is not a symmetric relation), there is no communication among roblets.

Roblet execution is temporized: during the initialization phase, a roblet declares the frequency at which its behavior should be executed. A roblet can also listen for incoming messages from bodymap or friends. The brain has a more traditional organization and is responsible for handling its own control flow; it is also responsible for hosting and controlling the bodymap. The bodymap notifies the arrival of incoming messages.

Since messages are sent asynchronously, 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. It is possible that a message gets lost, and this would be a problem in a model where messages flow only when there is something to be notified (interrupt-based); however, roblet communicates continuously with the brain whether there are changes or not, in the very same way as body organs always report to the brain.
Messages are represented as XML trees that are collected by the bodymap in a bigger tree. The advantage of this approach is that the entire status of the body is represented by a single tree. Therefore the brain has a homogeneous data structure to work with.

The framework has been implemented in C# and runs on the .NET framework [NET], as well as on Mono [MNP], and .NET compact framework (available on Windows CE based platforms).

The following class implements the heartbeat roblet, which simply notifies that it is alive:

```csharp
public class HeartBeatMessage : RobletMessage {
    public long beat;
    public HeartBeatMessage() {
        beat = DateTime.Now.Ticks;
    }
}
[OutputMessage(typeof(HeartBeatMessage))]
public class HeartBeat : RobletBase {
    public override void Initialize() {
        Frequency = 0.5;
    }
    public override void Run() {
        SendMessage(new HeartBeatMessage());
    }
}
```

In this example the roblet sends a heartbeat message every 30 seconds. The message is defined as a standard .NET class that will be serialized into XML by the framework inside the `SendMessage`. 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. The framework is responsible for reading the annotations from types using reflection programming interface.

Roblets may be asked for a message describing its interface; again the framework generates such an interface by reading annotation on types. A roblet may send messages to other roblets only if these are labeled with a `Friend` attribute indicating the type of the roblet. Communication is performed by using UDP messages.

**Reusing Roblets**

The design of Robotics4.NET has shown several interesting properties of the applications based on it. In particular, we noticed that the coupling of software modules has been reduced when we moved applications to our framework.

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 PC 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 a robot, it can be thought as if it is, 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 software organization with very little coupling.

We found several aspects of the framework that help decoupling software modules in robotic systems. In general software coupling may reduce software reuse, thus we found that being able to have a loosely coupled system may help to reuse software modules across different robotic systems.

The framework relies on XML messages for communication, as a consequence each roblet and the brain may avoid using the same set of types. As a matter of fact it is possible to write roblet or software that even avoids using XML serialization to deal with messages. In this respect we decouple modules by not sharing types.

If we compare roblets with body organs it is easy to notice 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 quite 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 leading again to a loosely coupled architecture.

We already exploited these characteristics to design software that is running on two completely different robotic architectures: we reused two software modules on ER1, a robot based on the robotic kit by Evolution Robotics [EVR], and on R2D2, an experimental robotic platform developed at our department [R2D2] (Whose appearance and functionality has been inspired from the popular Star Wars saga’s droid).

Related Work

The time for robotic frameworks has come: robotic hardware is getting cheaper and affordable, so now we need software which is powerful and expressive enough to develop services on these platforms. There are other robotic frameworks that provide suitable software infrastructures: the OROCOS project (Open Robot Control Software) [ORC], the MARIE project [MAR], and the Player/Stage project [PSP].

ORCOS has a highly modularized structure, essentially driver oriented, where it is easy to replace sensors and actuators simply by replacing the modules responsible for reading from these sensors or to control the actuators. However, the architecture is too fine grain, thus leading to a software design where the robot body is modelled by connecting several atomic components. Therefore the resulting software is rather coupled to a particular geometry of a robot. Besides, with Robotics4.NET we can avoid this by providing a suitable abstraction into a set of roblets that hide a certain amount of details.
Increasing decoupling in a framework for programming robots

MARIE provides a software infrastructure to develop adapters to wrap existing software and to make it work with other software modules. This approach has no control about the granularity of the software neither it supports software reusability. Nevertheless it helps to reuse existing software that is a good thing.

Player/Stage [IRO] is a software framework centered on the notion of “abstract device”. Like in OROCOS the application of a standard approach to software decomposition, the framework relies on a fine grain model that still separates sensors and actuators leading to software systems that inherently depend on the robot structure.

Conclusions

In this paper we presented Robotics4.NET, a framework for programming robots. We discussed how the framework design has been inspired to the notion of body that is the guiding principle in designing software based on the framework. We observed how this approach provides a loosely coupled model, whose granularity allows to define abstractions in the definition of the body shielding the brain software from the details of the underlying hardware.

The framework is still under development and it is used on several different projects to demonstrate how it can adapt to different domains.

Bibliography

[MNP] http://www.mono-project.com/Main_Page/
[R2D2] http://rotor.di.unipi.it/