Survey of Robot Programming Languages

Seminar Report

Submitted in partial fulfilment of the requirements for the degree of

Master of Technology

by

Anirban Basumallik
Roll No : 09305008

under the guidance of

Prof. Kavi Arya

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

April 2010
Abstract

At the present moment the field of Robotics is extremely varied. The nature of tasks that robots can perform are very distributed and hence controlling or programming different robots need different methods. Many vendors have tried to provide a common platform by abstracting out the differences. In this report we carry out a survey of the different Robot Programming Languages available in the market and assess their pros and cons. Finally, we come up with the needs of a desirable robotics platform that can ease the task of programmers in programming a class of robots more succinctly and effectively.
Contents

1 Introduction 3
  1.1 Robot Programming Platform 3
  1.2 Need for a Platform 3

2 Microsoft Robotics Developer Studio 3
  2.1 DSS 4
  2.2 CCR 5
  2.3 VPL 6
  2.4 VSE 7
  2.5 Others 8
  2.6 Pros, Cons and buzz 8

3 Player/Stage 9
  3.1 Player Goals and Design 9
  3.2 Stage simulation environment 10
  3.3 Working 11
  3.4 Pros, Cons and buzz 12

4 URBI 12
  4.1 Design philosophy 12
  4.2 URBI Technology 13
  4.3 Pros, Cons and buzz 14

5 OROCOS 15
  5.1 Design philosophy 15
  5.2 Pros, Cons and buzz 15

6 Comparative Study 16

7 General Requirements of a Platform 17

8 Conclusion 18

References 19

Appendices 20

A MSRDS Example 20

B Player/Stage Example 28
1 Introduction

This report primarily studies two major robot programming platforms. The first is Microsoft Robotics Developer Studio and the other one is the Player/Stage/Gazebo platform. Their general architecture, popularity in the industry, advantages and disadvantages have been closely studied. There is a brief report on two other platforms, namely, ORoCoS and URBI, which are promising and upcoming platforms. We then attempt to outline the qualities of a desirable robot programming language and suggest how we might go about building it.

1.1 Robot Programming Platform

What we mean by a Robot Programming Platform is:-

- Uniform programming syntax and commands
- Large support for hardware
- Abstraction from differences in hardware
- Availability of libraries for basic and common robotic tasks like joint control, image processing, and navigation.

Apart from these, the facilities for debugging through simulation and re-usability of components should be also available.

1.2 Need for a Platform

The need for a Robot Programming Platform arises from the fact that a lot of money needs to be invested in the software part of a Robotics project. Statistically [10], 80% of a project cost goes into software development. The main aim of a Robot Programming Platform is to reduce the project cost by reducing the software development cost. Rather the cost can be redirected to software development in the domain of algorithms and Artificial Intelligence. Having understood the basic needs behind a robotic platform let us now move into studying our first case study, Microsoft Robotics Developer Studio.

2 Microsoft Robotics Developer Studio

Microsoft launched their own Robotic programming platform, Microsoft Robotics Developer Studio (alias MSRDS) which can address a lot of the issues prevalent in the robotics industry. Powered by the strength of the C# programming language and the rich .NET framework, the MSRDS is a good platform for users. With a well laid out architecture, separate co-ordination library, and user-friendly GUI and simulation support, it makes this product quite a complete package on its own.

MSRDS comprises of the following:-

- **DSS**: Decentralised Software Services
- **CCR**: Concurrency and Co-ordination Runtime
- **VPL**: Visual Programming Language
- **VSE**: Visual Simulation Environment
2.1 DSS

DSS [5] stands for Decentralised Software Services. It is a lightweight runtime environment that sits on top of the CCR. It is a state-oriented service model that helps in developing scalable and high performance applications by integrating features like, isolation of services, composition of services, event driven notifications, message passing, and structured state modifications. It combines the notion of REpresentational State Transfer (REST) and Decentralised Software Services Protocol (DSSP) along with HTTP as the basis of interaction amongst services. DSSP is a Simple Object Access Protocol (SOAP) based protocol which provides a clean state transfer model which can manipulate states and an event model driven by state changes.

DSS [5] provides robustness by providing advanced error-handling features. Partial failures in sub-part of a system can lead to a complete halting of the whole system. Two types of isolation is needed for a robust system. Data needs to be isolated, as corrupted data can corrupt a service and execution isolation is needed, as one service depending on another service for execution may become unresponsive. A robust system always demands loosely coupled components. Such a system has additional demands of identifying, locating, synchronising, and composing them into one runnable unit. Unlike traditional systems which run as a single unit, DSS provides arrangements for creating, managing, and deploying loosely coupled systems. It has definitions of dependency of services which are verified at the initiation of the service. Having so many individual components running in parallel just adds to the problem of debugging. Locating the source of errors becomes a challenging task. Thus, the notion of observability has been integrated into the core of DSS by assigning a unique identifier for each service and observing the state changes that each service goes through as a part of its internal execution as well as external interaction.
2.2 CCR

**CCR** [4] stands for *Concurrency and Coordination Runtime*. It addresses the need to manage asynchronous operations, deal with concurrency, exploit parallelism and deal with partial failure. It encourages the design of loosely coupled components which can be developed independently and make minimal assumptions about other components in the runtime. This approach changes

---

**Service** A service [6] is the basic building block of DSS. A service is an abstraction over computation, it is a logical module. An entity or service has the following:-

- *Service Identifier*: Unique identifiers for locating an instance of a service. They are just identifiers and does not convey information about context or nature.

- *Contract Identifier*: These are condensed descriptions of implementations describing the behaviour. These are used to generate proxy dlls to which other services link against rather than directly linking to them.

- *State*: The state represents the internal states/current status of the service.

- *Partners*: The system maintains a list of partners who are dependent on the service or on whom the service depends for its proper functioning.

- *Handlers*: They handle the subscriptions and takes actions on an event occurring.

- *Main Port*: The main port is the central pipe of communication with all external entities. It is a CCR Port.

**Partnering of Services** Services declare each other as partners to work together. Partnering terms also exists which define the extent to which one partner can interact with the other. The following operations are available to interact with one another:-

- Create Services
- Query Services
- Manipulate services
- Subscribe to services
- Terminate Services
how the user thinks of a program from the designing process, dealing with concurrency, failure and isolation in a consistent way. Applications which use CCR need to coordinate through messages, deal with complex failure scenarios, or in other words has to deal with asynchronous programming. This model is also very similar to how heterogeneous hardware and network is built. It is operative in a way such that the whole system is responsive, scalable and efficient in dealing with failure. Some of the major features to mention about it are:-

- It is basically a dll, accessible from any language targeting the .NET Common Language Runtime (CLR).
- It has a data dependent schedule.
- It abstracts out the need to create threads or use semaphores.
- It acts as the glue between the runtime and the services.
- It takes care of load balancing, scheduling and coordinating.

Communication All the services interact through protocols and by exchanging messages. Network protocols are centered around http at the application layer and tcp in the network layer. The network protocols are e-secure. The system is also capable of communicating with enterprise web service architectures. The communication port maintains two different queues. One is for the data and the other is for the list of recipients. High level structures like Arbiter makes complicated queue logic look very simple.

Messages The messages exchanged are human readable, serialized into XML over the network. The content of the messages directly correspond to the impact of the message. They are of the following types: modification, query, get, drop. As a matter of statistics, which gives an idea about the efficiency of the system, on an Intel Dual Core, 3.0 GHz machine, 50,000 messages could be exchanged per second and about 3,000 messages per second could be exchanged over distant machines.

2.3 VPL

VPL [8] stands for Visual Programming Language. As the name suggests, it is a GUI based programming logic system. It is actually a graphical data-flow programming model. The exact logic of the program is represented as sequences of blocks with inputs and outputs, which are connected. In Figure 3, we can see a VPL program, which is more like a logic diagram than a program.

Example In Figure 3, we see a program that makes the computer to count from 1 to 10. The components are needed to be just dragged and dropped from a toolbar and connected as per the flow in which the user wants to program. In the program we see in Figure 3, the execution starts at the left-most red arrow at the Data block having a value of 1. This value is set to the Test variable of the next block and as indicated by the arrow marker. Then the flow goes through the Merge point and continues in two parallel flows. The top flow calculates a string variable whose value is “The number is 1” and then reads it out through the speakers. The other flow enters the If block and checks the condition specified. This time the condition is not satisfied, hence, the flow takes the Else route and enters the Calculate block which calculates Test + 1 ⇒ 1 + 1 ⇒ 2 and then enters the variable block which sets the value of the Test variable to 2. The flow again enters the Merge block and splits into two parts. One part speaks out “The number is 2” and the other part continues with the calculation.
With the help of the Visual Programming Language, programming becomes very easy. The ease with which the hardware can be referenced completely abstracts out the hardware complexity. The ease of use of high level functionalities like text-to-speech can be done very easily. At the same time, the ease provided here comes at the cost of extreme hardwork given in by the backend designers.

2.4 VSE

VSE stands for Visual Simulation Environment. The VSE provides the simulation to substitute hardware. Hardware has multiple disadvantages like high cost, difficulty of debugging and tough to be worked upon concurrently. Whereas simulation has a low barrier to entry and its very easy prototype and test out new ideas. At the same time simulation has a lack of noisy data i.e. assumes a perfect world and it is also tough to model a hardware absolutely perfectly.

Simulation Architecture  The simulation architecture comprises of the following as shown in Figure 4:-

- A editor for modelling and debugging
- The simulation engine service
- AGEIA physics engine
2.5 Others

- XNA Graphics Library for 3D rendering
- The display hardware

The simulation engine is also implemented as a DSS service. Just as the hardware would have connected to the platform, the simulation engine does so. Generally 3rd party softwares like Maya or 3DsMax is used to design models with accuracy.

The simulator has entities. Each entity represents real world objects like sensors, actuators, obstacles and others. Every entity has two aspects to it. One is the visual component and the other is the physics component. The physics component decides and simulates how the entities interact with each other in the simulator and the visual component decides as to how the entity would look in the simulator. To link the simulation entities with the corresponding services we use xml files, in which we specify the contract identifier and declare it as a partner as shown in Figure 5.

Figure 5: How the simulation entities interact with the corresponding services[7]

2.5 Others

Apart from the above mentioned features, the platform of MSRDS also provides many other small facilities which are of extreme importance. With such a huge system having a high number of loosely coupled components working in a distributed nature, it is of utmost importance to have a log of all the activities. MSRDS does provide with such logs which are also very important in debugging. At the same time when data is being transferred over the net, security of data can be a major factor. MSRDS also does provide that. Apart from these, it also provides detailed research diagnostics like CPU utilization, list of active and pending queues at each instant and such others.

2.6 Pros, Cons and buzz

MSRDS is a good development environment for designing, executing, and debugging highly scalable, concurrent and distributed robotics applications. It has got extremely efficient backend with a fantastic simulator and with the added advantage of the VPL, robotics applications have become very easy to compose. But at the same time, knowledge of coding is required for some significant work. The complexity of C#, modelling of objects and such others provide a barrier for entry to many people. It also doesn’t provide any special support for real-time applications. Microsoft is doing what it does best: marketing. By organising summits and
discussion opportunities in the academic domain and having the big robotic hardware designing firms in its support and confidence, one thing is assured that this product is here to stay and maybe also dominate.

3 Player/Stage

“All the world’s a stage, And all the men and women merely players.”

- William Shakespeare, As You Like It

The project derives its name from the above famous saying. The Player/Stage project began at the USC Robotics Research Lab in 1999. It has since been adopted, modified and extended by researchers from around the world. The Player/Stage project is the standard and the most popular robot programming platform in the open source community. It has got client side programming support for C/C++, Java, Tcl, and Python. It supports a variety of hardware and its modular architecture makes development easy.

3.1 Player Goals and Design

The design philosophy as perceived from the publications [3] and [11] and from usage of the software we identify the following:-

![Player Architecture](image)

**Figure 6: Player Architecture[9]**

**Player Core** Initially Player was designed as a *socket-based device server* which is connected to a collection of many devices and provided a TCP socket which can be connected to by applications of different languages. The languages which are available are C, C++, Tcl, Python, and Java. The socket based model provides location independence and it can be extended to a distributed nature. At the present moment (i.e. Player 2.0), the core model has been modified to a queue-based message passing system. Each driver has a single incoming queue and they can send messages on the queues of others. The core library is responsible for the message semantics and ensures the coordination of the message passing system. This is an improvement over the multiple heterogeneous queues that existed earlier. Apart from these, the addressing system of the hardwares has also been modified to entertain accurate addressing and providing arrangements for inter-server subscriptions. In Figure 7, a possible connection of different servers is shown.
3.2 Stage simulation environment

Device Model  Like the Linux philosophy, Player also deals with a device as a file. Hardwares are dealt with in a manner like operating systems. There is a common interface for all devices of the same type and separate drivers which glues the interface to the actual hardware.

Transport Layer  The transport layer does still have TCP sockets as the basic interface for communication but it has changed the logical entities on the two sides of the socket. On the driver side of the socket there are incoming and outgoing queues, whereas in the communication side of the socket, there has been a change introduced in the format of the packet. Initially it used to be packed C structures, but now it uses an open standard called eXternal Data Representation or XDR. It is an efficient, platform independent, format for common data-types.

3.2 Stage simulation environment

It is a simulation environment for Player that renders the scenarios in 2D. Though the new version has got 3D rendering abilities, it is basically a 2D environment with a concept of elevation. It is like a plugin that connects to the Player server just in the same manner as the hardware would have connected to it. The idea of Stage’s connectivity is displayed in Figure 8.
3.3 Working

In Figure 8, we see that the Stage simulator tricks the Player Server by substituting the hardware and cloning the drivers. Stage was designed with the multi-robot systems (MRS) in mind. Hence speed was more important than accuracy. Another motivation for designing Stage was to substitute hardware and reduce cost. It also gave a chance to experiment with new types of sensors which are not yet available as readymade hardware in the market. It also helps in studying multi robot behaviours, which would have been otherwise very costly.

Some of the major features of Stage [3] are:-

- **Optimal fidelity**: It looks more into performance than accuracy.
- **Linear Scaling with population**: The algorithms are such designed that they do not depend on the number of the robots.
- **Configurable device models**: The sensor models are modelled with a lot of flexibility and to conform as much closely as possible to the actual nature of the hardware. They are fully configurable as much as the original hardware provides.
- **Abstraction from Client code**: The sensors are available through the Player interface, hence the client code cannot distinguish from the real sensors or the simulated sensors.

![Figure 9: Player/Stage in action: An obstacle avoiding robot of Appendix B](image)

3.3 Working

The basic working mechanism of the Player/Stage platform is explained in Figure 8. For its proper functioning, there is a need to understand three file types:-

- **.world file**: It tells Player/Stage what things are available to put in the world. We describe our robot, any items which populate the world and the layout of the world.
• .inc file: They are also of the same format as that of the .world files, but they are generally used to define one entity only. These files are then included in the final .world files which represents the simulation.

• .cfg file: This is the most important file as this specifies which driver to use for which device and the location of the drivers.

Another concept that is needed to be understood, is that of a model. Everything is represented as model in the simulation. The map, objects, obstacles, robots and all are models. Every model has a name and a host of parameters which defines its physical properties like colour, draggable, grippable, and others. There are a lot of built-in models representing the basic needs of robotics like sensors and actuators. Then comes the external code. As mentioned before, the user can code his/her algorithm in various languages like C, C++, Java, and Python according to the user’s wish.

3.4 Pros, Cons and buzz

The logical layout of the architecture and a maintenance of close configurability with its hardware counterparts, gives a close feeling of using hardware rather than software. The ease of use and the simplicity alongwith the usage of our favourite programming language makes it a very popular platform amidst academicians and hobbyists. But the absence of robustness, missing constraints of real-time demands, inability to be ported cleanly on Windows platform and the difficulty of installation causes a hindrance in its popularity. Whatever, the platform of Player/Stage is the de facto standard in the domain of robotics in the open source community.

4 URBI

This is another proprietary Robot Programming Language. URBI stands for Universal Real-time Behaviour Interface and is developed by a company named Gostai, dedicated to Artificial Intelligence applied to Robotics, based in France. It is basically a new language all together and the company defends their stand as to why a new language is needed for present day robotics applications. Their mission is to provide the users with the best universal robotic platform. The product has been developed by leading Universities in the field of Artificial Intelligence since 2003.

4.1 Design philosophy

URBI’s observation is that we are entering the robotic age and still there is incompatibility in terms of software for the available hardware of robotics. It is much like the problem of having different PC architectures faced by the computer industry in its nascent days. Hence it is their effort to bridge the gap by providing a common platform for robotic applications. The product [2] has been designed with the following principles in mind:-

Flexibility The platform should work with any robot, operating system or programming environment.

Modularity It follows an architecture in which components are modular in nature, hence built and maintained separately. They are all executed together in the platform.
4.2 URBI Technology

Powerful Abstractions  Robotics is a very complex domain and at the same time there exists a lot of redundancy which needs to be abstracted out to increase efficiency and elegance.

Simplicity  Another principle which URBI strongly follows is that none of the above things should be tough to understand or use. Things should be so easy such that hobbyists or kids can have a quick kickstart.

4.2 URBI Technology

URBI is quite an innovation in the programming world as it is not just a programming language (urbiScript) but also has a component architecture (UObject) and also several graphical programming tools (urbiStudio). Its integration of control over parallelism in the syntax of the language is a very noble approach. All this along with a familiar C++ like syntax, makes it a very attractive platform.

Lets have a look at the two pillars of their technology:-

- **URBI Script**: This forms the heart of the technology. It brings totally new features and simplicity in the issues of parallelism and event-based programming. It is also fully interfaced with other languages like C++, Java, Matlab, Python, and Ruby. It doesn’t allow full control of flow to these traditional languages, but just allows it to issue URBI commands using the traditional programming constructs. So basically, it is the URBI server which keeps control over the flow of execution and executes the commands as they keep coming from the user code. Parallelism and events are directly controlled and also visible in the syntax of the urbiScript. Some real-time aspects are also controlled directly by the urbiScript. For instance, if a task has to be spread over a specific time span, then it can be directly specified in the code, just beside where the task is initiated. Figures 10 and 11 exhibit a few of these features.

![Figure 10: Event-Handling and Parallelism in urbiScript](image)

In Figure 10, we see both the features of event-handling and parallelism being present. The event-handling keyword ‘whenever’ declares this piece of code to be invoked when the event ‘ball.visible’ (i.e. ball is within camera’s view field) occurs. The ‘headPan’ variable represents the servo motor which controls the movement of the robot’s head about the vertical axis. Similarly, the ‘headTilt’ variable represents the servo motor which controls the robot’s head movement about the horizontal axis. The parallelism feature is represented in the code by the ‘&’ operator. By using it between the two codes, the two commands are executed paralelly and we will be able to see that the head of the robot moves along the diagonal towards the ball. Had it been executed serially, then the
robot would have first aligned its head along the x-axis of the ball then aligned itself along
the y-axis of the ball.

![urbiScript](image)

Figure 11: Advanced features in urbiScript[2]

In Figure 11, we see three different pieces of code. In the first code, we see some
constraints being attached to the codes. The first line mentions that the ‘neck’ motor’s
value will be set to ‘10’ over a ‘timespan’ of ‘450ms’. The second line specifies that the
‘leg’ motor’s value is set to ‘-45’ degrees and the transition occurs at the speed of ‘7.5’
units. The third line says that ‘tail’ motor’s value will be ‘14’ and the transition will
occur over a sin wave like motion of amplitude ‘45’ units, and the transition will take place
over a period of 4s.

The second part of Figure 11, handles an anomalous behaviour of parallelism, where the
same variable may be demanded to be modified and set two different values. Such cases are
handled by the ‘blend’ property which defines what to do, when such a situation comes.
For example, in this figure, it is mentioned that it will ‘add’ thus the value of ‘x’ becomes
‘4’ as the effect is added.

The third part of Figure 11, displays some features of code tagging. A part of code, which
are inside the curly braces, is tagged as ‘mytag’. That part of code can be referred to from
any part of the program by the tag, and commands like stop (to stop the code), freeze
(to pause the code), and unfreeze (to start the frozen code) can be executed upon them.

- **UObject Architecture**: The UObjects are very similar to C++ objects. Infact C++ objects
  can be very easily converted to UObject. There was a need to have a different object
  architecture as they wanted the objects to conform to their overall principle of simplicity
  and have abstractions over network complexities. The principle can be extended to any
distributed object architecture and it bridges the gap between other component architectures like CORBA. In UObject architecture it is also possible to add new methods at
runtime to a live object.

### 4.3 Pros, Cons and buzz

The main advantage of the product is the ease with which parallelism and event handling can
be done. Some time constraints can also be very easily controlled by the user. The disadvantage
would include the pain of having to learn a new language. Even if it does have the facility to
integrate with other languages, but it doesn’t give the full control to those languages, rather
they have to generate URBI commands to the URBI server. It is quite a popular product for the hardware that it supports, and its popularity may increase from May, 2010 onwards as it is going to be open source from then onwards.

5 OROCOS

OROCOS stands for Open RObot COntrol Software. Their aim is to build a generic modular framework for robot and machine control. They just provide libraries for building but does not strictly provide a platform. They give stress on accuracy rather than on computation.

![Figure 12: OROCOS libraries[1]](image)

5.1 Design philosophy

Their design philosophy [1] is centered around four libraries, namely:-

- **RealTime Toolkit (RTT)**: This library ensures realtime constraints.

- **Orocos Components Library (OCL)**: This library is used to design components which represents real life entities like sensors and actuators. It uses the KDL.

- **Kinematics and Dynamics Library (KDL)**: It is an application independent framework for modelling and computation of kinematic chains, such as robots, biomechanical human models, computer-animated figures, machine tools, etc.

- **Bayesian filtering library (BFL)**: It is a library which uses statistical models to induce noise into the models, which otherwise assumes a perfect world.

As seen in Figure 13, to build the base of an application, the RTT, BFL and KDL libraries are used. Then for the middle layer OCL is used, above which applications are developed and then used by the final user.

5.2 Pros, Cons and buzz

It is the only platform that takes deep care of realtime issues. Unfortunately, it is not a platform as a whole, hence most of the functionalities of networking, distribution of load, and hardware abstraction is left to the users. Yet, for those who want accurate results, and a perfect simulation, this is the best library to look out for.
6 Comparative Study

Let's present a comparative study [10] of the four main products, in Table 1, which have been discussed above. Thus we can see that each product has its own advantages, disadvantages and specialities. In MSRDS, the loosely coupled nature of the DSS library, the distributed nature and efficiency of the CCR library and high quality of simulation gives it an edge. The simplicity and the facility to use our favourite programming language makes Player/Stage a popular platform. The ease of programming in URBI, as many complexities of real life demands are abstracted out, makes it a very desirable platform. The parallelism handling feature of URBI makes it stand apart from the others. The intricacies and accuracy of the OROCOS library is a challenging task to build from scratch and it serves the purpose of applications which demand perfection. Having highlighted the dominating features of the above products, we would now like to move onto (Section 7) the description of a desirable robotic platform which has the best features of all the products mentioned above as well as some features totally missing from the above ones.

<table>
<thead>
<tr>
<th></th>
<th>MSRDS</th>
<th>Player/Stage</th>
<th>URBI</th>
<th>OROCOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Source</td>
<td>No</td>
<td>Yes</td>
<td>from May’10</td>
<td>Yes</td>
</tr>
<tr>
<td>Windows</td>
<td>Yes</td>
<td>Emulation</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Linux</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Nature</td>
<td>Platform</td>
<td>Platform</td>
<td>Platform</td>
<td>Library</td>
</tr>
<tr>
<td>Distributed</td>
<td>Yes</td>
<td>Partially</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>In-built Arm Control</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Simulation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Reusability</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Real-Time</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Comparative Study of Robot Programming Languages
7 General Requirements of a Platform

After having studied some of the most popular robotic platforms available and having used a few of them, we would like to factor out all the important features of these products and list the requirements of a complete robot programming platform. They are:-

- **Abstraction and Uniformity**: The foremost requirement of a platform is to provide uniformity of commands and syntax to control robots and lay down a complete abstraction over the hardware differences.

- **Scalability and Distributed Nature**: With the era of distributed computing coming ahead, it is of utmost importance to provide distributed behaviour in the platform or to ensure that it conforms to some standard which would help it to run on a distributed layer. The issue of scalability can be in general solved by providing a distributed nature but at the same time it needs to be ensured that the scalability should result in linearly scaled performance with increase in size of population.

- **Modularity and Re-usability**: Another very important issue is that of modularity. Since the real world components are modular by nature, and the hardware components are also generally produced by different vendors, the platform should also deal with them separately. This would help in isolating the errors and would also ease out development as the developers would be only concentrating on a module and deal with it independently of the platform. This would also increase re-usability, as a component developed once can be used perpetually.

- **Parallelism and Synchronisation**: In robotic applications, it is often required to do two or more tasks in parallel, at the same time not conflicting with each other. Thus parallelism and synchronisation should not only be available but also should be well abstracted such that the user does not have to think about the coding complexities of threads and semaphores but should just be concerned with the top level logic of dealing with them.

- **Real-Time Constraints**: Since robotic applications are all real world applications, handling of real time constraints should be always expected from the platform.

- **Error Handling and Debugging**: One of the most important issues related to any form of software is the facility to debug it. Having a loosely coupled system distributed over a network, makes the job of debugging tougher. Having a simulation environment does ease the pain of debugging a lot but debugging in the simulation itself can be quite a challenging task.

- **In built support for fundamental robotic activities**: Fundamental robotic activities like image processing, robotic arm control and such other facilities should be readily available such that the user can concentrate on the algorithm or artificial intelligence component of the system and not waste much time with the subtleties of the coding and handling well established fields.

- **Simplicity**: Above all, the features provided should be all very simple to learn and use and should not be very complex by itself.
8 Conclusion

Thus we have had a study of some interesting products in the world of robotic programming. We have also had a taste of development using two of them. The pros and cons of various other products in the market have been studied and presented in a manner that brings out the needs of the present situation. Finally, a list of the general requirements of a robot programming platform has been suggested, which if implemented can meet all the varying demands of robotic development with ease.
References


Appendices

A MSRDS Example

Here is a code whose screenshot is given in Figure 1.

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using Microsoft.Ccr.Core;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.ServiceModel.Dssp;
using Microsoft.Dss.ServiceModel.DsspServiceBase;
using W3C.Soap;
using submgr = Microsoft.Dss.Services.SubscriptionManager;

using Microsoft.Robotics.PhysicalModel;


// Following are used for accessing timer to create timeout to call a function to display sensor data
using System.Linq;
using System.Text;
using System.Timers;

namespace LineFollowerTrialEnv1
{
    [Contract(Contract.Identifier)]
    [DisplayName("LineFollowerTrialEnv1")]  
    [Description("LineFollowerTrialEnv1 service (no description provided)")]  
    class LineFollowerTrialEnv1Service : DsspServiceBase
    {
        // <summary>
        // Service state
        // </summary>
        [ServiceState]
        LineFollowerTrialEnv1State _state = new LineFollowerTrialEnv1State();
    }
}
```
/// <summary>
/// Main service port
/// </summary>
[ServicePort("/LineFollowerTrialEnv1", AllowMultipleInstances = true)]
LineFollowerTrialEnv1Operations _mainPort = new
LineFollowerTrialEnv1Operations();

[SubscriptionManagerPartner]
submgr.SubscriptionManagerPort _submgrPort = new submgr.SubscriptionManagerPort();

/// <summary>
/// SimulationEngine partner
/// </summary>
    Identifier, CreationPolicy = PartnerCreationPolicy.
    UseExistingOrCreate)]
engine.SimulationEnginePort _simulationEnginePort = new
engine.SimulationEnginePort();

Port<RaycastResult> _rayCastPort = new Port<RaycastResult>();

/// <summary>
/// Service constructor
/// </summary>
public LineFollowerTrialEnv1Service(DsspServiceCreationPort creationPort)
    : base(creationPort)
{
}

/// <summary>
/// Service start
/// </summary>
protected override void Start()
{

    // Add service specific initialization here
    //
    base.Start();
    // Orient sim camera view point
    SetupCamera();
    // Add objects (entities) in our simulated world
    PopulateWorld();
///The following code entered for displaying sensor data periodically
System.Timers.Timer aTimer = new System.Timers.Timer();
aTimer.Elapsed += new ElapsedEventHandler(
    displaySensorData);
// Set the Interval to 5 seconds.
aTimer.Interval = 5000;
aTimer.Enabled = true;
}

void displaySensorData(object source, ElapsedEventArgs e)
{
    Console.WriteLine("Hello World!");

    RaycastResult verticalResults;
    _rayCastPort.Test(out verticalResults);
    if (verticalResults != null)
    {
        foreach (RaycastImpactPoint impact in verticalResults.ImpactPoints)
        {
            Console.WriteLine(impact.Position.ToString());
        }
    }
}

bool HasError<T>(PortSet<T, Fault> sensorOrFault)
{
    Fault fault = (Fault)sensorOrFault;
    if (fault != null)
    {
        LogError(fault.ToException());
        return true;
    }
    else
    return false;
}

private void SetupCamera()
{
    // Set up initial view
    CameraView view = new CameraView();
    view.EyePosition = new Vector3(-0.85f, 0.83f, -0.29f);
    view.LookAtPoint = new Vector3(-0.34f, 0.69f, -0.14f);
    SimulationEngine.GlobalInstancePort.Update(view);
}

private void PopulateWorld()
{
AddSky();
AddGround();
AddTable(new Vector3(1, 0.5f, -2));
AddLine(new Vector3(1, 0, 1));
AddLine2(new Vector3(2, 0, 1));
AddPioneer3DRobot(new Vector3(1, 0.1f, 0));

#region CODECLIP 03-3
void AddSky()
{
    // Add a sky using a static texture. We will use the sky texture
    // to do per pixel lighting on each simulation visual entity
    SkyDomeEntity sky = new SkyDomeEntity("skydome.dds", "sky_diff.dds");
    SimulationEngine.GlobalInstancePort.Insert(sky);

    // Add a directional light to simulate the sun.
    LightSourceEntity sun = new LightSourceEntity();
    sun.State.Name = "Sun";
    sun.Type = LightSourceEntityType.Direction;
    sun.Color = new Vector4(0.8f, 0.8f, 0.8f, 1);
    sun.Direction = new Vector3(0.5f, -0.75f, 0.5f);
    SimulationEngine.GlobalInstancePort.Insert(sun);
}
#endregion

#region CODECLIP 03-4
void AddGround()
{
    // create a large horizontal plane, at zero elevation.
    HeightFieldEntity ground = new HeightFieldEntity("simple ground", // name
        "blackSurface.dds", // texture image
        new MaterialProperties("ground",
            0.2f, // restitution
            0.5f, // dynamic friction
            0.5f) // static friction
    );
    SimulationEngine.GlobalInstancePort.Insert(ground);
}
#endregion

void AddTable(Vector3 position)
{
    // create an instance of our custom entity
    TableEntity entity = new TableEntity(position);
}
// Name the entity
entity.State.Name = "table:" + Guid.NewGuid().ToString();

// Insert entity in simulation.
SimulationEngine.GlobalInstancePort.Insert(entity);
}

#region CODECLIP 03–5
void AddLine(Vector3 position)
{
    Vector3 dimensions =
    new Vector3(0.5f, 0.01f, 0.05f); // meters

    // create simple movable entity, with a single shape
    SingleShapeEntity box = new SingleShapeEntity(
        new BoxShape(
            new BoxShapeProperties(
                10, // mass in kilograms.
                new Pose(), // relative pose
dimensions)), // dimensions
        position);

    box.State.MassDensity.Mass = 0;
    box.State.MassDensity.Density = 0;

    // Name the entity. All entities must have unique names
    box.State.Name = "box";

    // Insert entity in simulation.
    SimulationEngine.GlobalInstancePort.Insert(box);
}

void AddLine2(Vector3 position)
{
    Vector3 dimensions =
    new Vector3(0.5f, 0.01f, 0.05f); // meters

    // create simple movable entity, with a single shape
    SingleShapeEntity box = new SingleShapeEntity(
        new BoxShape(
            new BoxShapeProperties(
                10, // mass in kilograms.
                new Pose(new Vector3(0, 0, 0), new Quaternion(0.0
f, 0.383f, 0.0f, 0.9239f))), // relative pose
dimensions)), // dimensions
        position);

    box.State.MassDensity.Mass = 0;
    box.State.MassDensity.Density = 0;
// Name the entity. All entities must have unique names
box.State.Name = "box2";

// Insert entity in simulation.
SimulationEngine.GlobalInstancePort.Insert(box);

#region

void AddPioneer3DXRobot(Vector3 position)
{
    Pioneer3DX robotBaseEntity = CreateMotorBase(ref position);

    // Create Laser entity and start simulated laser service
    LaserRangeFinderEntity laser = CreateLaserRangeFinder();
    // insert laser as child to motor base
    robotBaseEntity.InsertEntity(laser);

    // Finally insert the motor base and its child sensor
    // to the simulation
    SimulationEngine.GlobalInstancePort.Insert(robotBaseEntity);
}

private LaserRangeFinderEntity CreateLaserRangeFinder()
{
    // Create a Laser Range Finder Entity.
    // Place it 10cm above base CenterofMass.
    LaserRangeFinderEntity laser = new LaserRangeFinderEntity(
        new Pose(new Vector3(0, 0.10f, 0)));
    laser.State.Name = "P3DLaserRangeFinder";
laser.LaserBox.State.DiffuseColor = new Vector4(0.25f, 0.25f, 0.8f, 1.0f);

    // Create LaserRangeFinder simulation service and specify
    // which entity it talks to
    lrf.Contract.CreateService(
        ConstructorPort,
        CreateEntityPartner(

    laser.Register(_rayCastPort);

    return laser;

#endregion
private Pioneer3DX CreateMotorBase(ref Vector3 position)
{
    // use supplied entity that creates a motor base
    // with 2 active wheels and one caster
    Pioneer3DX robotBaseEntity = new Pioneer3DX(position);

    // specify mesh.
    robotBaseEntity.State.Assets.Mesh = "Pioneer3dx.bos";
    // specify color if no mesh is specified.
    robotBaseEntity.ChassisShape.State.DiffuseColor = new Vector4(0.8f, 0.25f, 0.25f, 1.0f);

    // the name below must match manifest
    robotBaseEntity.State.Name = "P3DXMotorBase";

    #region CODECLIP 04−1
    // Start simulated arcos motor service
    drive.Contract.CreateService(ConstructorPort,
        CreateEntityPartner(
    );
    #endregion
    return robotBaseEntity;
}

/// <summary>
/// Handles Subscribe messages
/// </summary>
/// <param name="subscribe">the subscribe request</param>
[ServiceHandler]
public void SubscribeHandler(Subscribe subscribe)
{
    SubscribeHelper(_submgrPort, subscribe.Body, subscribe.
        ResponsePort);
}

[DataContract]
public class TableEntity : MultiShapeEntity
{
    /// <summary>
    /// Default constructor.
    /// </summary>
    public TableEntity() { }
}
A MSRDS EXAMPLE

/// <summary>
/// Custom constructor, programmatically builds physics primitive shapes to describe
/// a particular table.
/// </summary>
/// <param name="position">public TableEntity(Vector3 position)
{
    State.Pose.Position = position;
    State.Assets.Mesh = "table_01.obj";
    float tableHeight = 0.65f;
    float tableWidth = 1.05f;
    float tableDepth = 0.7f;
    float tableThinkness = 0.03f;
    float legThickness = 0.03f;
    float legOffset = 0.05f;

    // add a shape for the table surface
    BoxShape tableTop = new BoxShape(
        new BoxShapeProperties(30,
            new Pose(new Vector3(0, tableHeight, 0)),
            new Vector3(tableWidth, tableThinkness, tableDepth))
    );

    // add a shape for the left leg
    BoxShape tableLeftLeg = new BoxShape(
        new BoxShapeProperties(10, // mass in kg
            new Pose(
                new Vector3(-tableWidth / 2 + legOffset, tableHeight / 2, 0),
                new Vector3(legThickness, tableHeight + tableThinkness, tableDepth))
        );

    BoxShape tableRightLeg = new BoxShape(
        new BoxShapeProperties(10, // mass in kg
            new Pose(
                new Vector3(tableWidth / 2 - legOffset, tableHeight / 2, 0),
                new Vector3(legThickness, tableHeight + tableThinkness, tableDepth))
        );

    BoxShapes = new List<BoxShape>();
    BoxShapes.Add(tableTop);
    BoxShapes.Add(tableLeftLeg);
    BoxShapes.Add(tableRightLeg);
}
public override void Update(FrameUpdate update)
{
    base.Update(update);
}

B Player/Stage Example

Let us now go through an example which will help in understanding the features better:

The robot’s .inc file. This file [9](named bigbob.inc) contains the description of the robot as a two dimensional object identified by points of a polygon. The sensor’s nature, positions, count and other physical parameters are also specified here. Here is the file:-

#Bigbob’s sonars
define bigbobs_sonars ranger
{
    # number of sonars
    scount 4

    # define the pose of each transducer [xpos ypos heading]
    spose[0] [ 0.75 0.1875 0] #fr left tooth
    spose[1] [ 0.75 -0.1875 0] #fr right tooth
    spose[2] [ 0.25 0.5 30]       # left corner
    spose[3] [ 0.25 -0.5 -30]    # right corner

    # define the field of view of each transducer
    #[range_min range_max view_angle]
    sview [0.3 2.0 10]

    # define the size of each transducer
    # [xsize ysize] in meters
    ssize [0.01 0.05]
}

# bigbob’s blobfinder
define bigbobs_eyes blobfinder
{
    # number of colours to look for
    colors_count 2

    # which colours to look for
    colors ["orange" "DarkBlue"]

    # camera parameters
    image [160 120]  #resolution
}

# bigbob’s laser
define bigbobs_laser laser
(
    # range_min 0.0
    # distance between teeth in metres
    range_max 0.25

    # does not need to be big
    fov 20

    # pose [0.625 0.125 0.1 270]
    size [0.025 0.025 0.01]
)

# bigbob’s body
define bigbob position
(
    # actual size
    # size [1.25 1 1]
    size [0.625 0.5 0.5]

    # Bigbob’s centre of rotation is offset from its centre of area
    origin [0.125 0 0 0]

    # estimated mass in KG
    mass 15.0

    # the shape of Bigbob
    block
    ( points 6
        point[5] [0 0]
        point[4] [0 1]
        point[3] [0.75 1]
        point[2] [1 0.75]
        point[1] [1 0.25]
        point[0] [0.75 0]
        z [0 1]
    )

    block
    ( points 4
        point[3] [1 0.75]
        point[2] [1.25 0.75]
        point[1] [1.25 0.625]
        point[0] [1 0.625]
    )
\[ z \left[ 0 \ 0.5 \right] \]

\)

block
(

points 4
point[3] [1 0.375]
point[2] [1.25 0.375]
point[1] [1.25 0.25]
point[0] [1 0.25]
\)

# differential steering model
drive "diff"

# sensors attached to bigbob
bigbobs_sonars()
bigbobs_eyes()
bigbobs_laser()

obstacle\_return 1 # Can hit things.
laser\_return 1 # reflects laser beams
ranger\_return 1 # reflects sonar beams
blob\_return 1 # Seen by blobfinders
)

The map.inc file specifies the map, on which the robot will move. It is as follows:-
define map model
(

# sombre, sensible, artistic
color "black"
# most maps will need a bounding box
boundary 1
gui\_nose 1
gui\_grid 1
#ranger\_return 1 # reflects sonar beams
gui\_move 0
gui\_outline 0
fiducial\_return 0
gripper\_return 0
laser\_return 1
)

The empty.world [9] contains all the information of the environment. It also includes the .inc files defined above. It is as follows:-
include "map.inc"
include "bigbob.inc"
#include "sick.inc"

# size of the whole simulation
size [15 15]
# configure the GUI window
window
{
    size [650.0 650.0] # in pixels
    # 650/15 rounded up a bit
    scale 43 # in pixels/meter
    #show_data 1
}

# load an environment bitmap
map
{
    bitmap "bitmaps/cave.png"
    pose [0 0 0 0]
    size [15 15 1.60]
}

bigbob
{
    name "bob1"
    pose [-5 -6 0 0]
    color "red"
}

The empty.cfg file [9] specifies the drivers which the Player Server will use. It is as follows:-

driver
{
    name "stage"
    plugin "stageplugin"
    provides ["simulation:0"]
    # load the named file into the simulator
    worldfile "empty.world"
}

driver
{
    name "stage"
    provides ["6665:position2d:0" "6665:blobfinder:0" "6665:laser:0" "6665:sonar:0"]
    model "bob1"
}

The Player Server needs to be invoked after these four files have been defined. To do so, use the following command:-

player empty.cfg
This will not only initiate the Player server but also the Stage simulation window. The robot can be located in it in red colour.

The user code (code.cpp) which controls the robot is as follows:-

```cpp
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <libplayerc++/playerc++.h>

using namespace PlayerCc;
using namespace std;

void AvoidObstacles(double *forwardSpeed, double *turnSpeed, SonarProxy &sp)
{
    // will avoid obstacles closer than 40cm
    double avoidDistance = .4;
    // will turn away at 10 degrees/sec
    int avoidTurnSpeed = 10;
    // left corner is sonar no. 2
    // right corner is sonar no. 3
    if (sp[2] < avoidDistance)
    {
        *forwardSpeed = 0;
        // turn right
        *turnSpeed = (-1)*avoidTurnSpeed;
        return;
    }
    else if (sp[3] < avoidDistance)
    {
        *forwardSpeed = 0;
        // turn left
        *turnSpeed = avoidTurnSpeed;
        return;
    }
    else if ((sp[0] < 2*avoidDistance) || (sp[1] < 2*avoidDistance))
    {
        // back off a little bit
        *forwardSpeed = -0.3;
        if (sp[0] < sp[1]) // If left sensor is closer then right sensor
            *turnSpeed = -(rand() %70+20); // Take a right turn randomly ranging between 20 and 90 degrees
        else
            *turnSpeed = rand() %70+20; // Take a random left turn
        return;
    }
    return; // do nothing
}
```
int main(int argc, char *argv[])
{
    // Enabling Proxies
    PlayerClient robot("localhost");
    Position2dProxy p2dProxy(&robot, 0);
    SonarProxy sonarProxy(&robot, 0);
    BlobfinderProxy blobProxy(&robot, 0);
    LaserProxy laserProxy(&robot, 0);

    // enable motors
    p2dProxy.SetMotorEnable(1);
    // request geometries
    p2dProxy.RequestGeom();
    sonarProxy.RequestGeom();
    laserProxy.RequestGeom();
    // blobfinder doesn't have geometry

    // some control code
    double forwardSpeed, turnSpeed;
    srand(time(NULL));
    while (true)
    {
        // read from the proxies
        robot.Read();

        forwardSpeed = .7;
        turnSpeed = 0;

        // call a function to avoid obstacles
        AvoidObstacles(&forwardSpeed, &turnSpeed, sonarProxy);

        // set motors
        p2dProxy.SetSpeed(forwardSpeed, dtor(turnSpeed));
        sleep(1); // wait for one second
    }
    return 0;
}

This code should be compiled using:

g++ -o code 'pkg-config --cflags playerc++' code.cpp 'pkg-config --
libs playerc++'

Then the code should be executed using './code' command. This should connect to the Player Server and the robot should start moving according to this algorithm.